Ember.js

Conheça o framework
para aplicações web ambiciosas

Casa do Código

CLAIRTON RODRIGO

ISBN

Impresso e PDF: 978-85-94188-03-8

EPUB: 978-85-94188-04-5

MOBI: 978-85-94188-05-2

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Este produto não é afiliado ao projeto Ember. Ember é uma marca registrada por Tilde Inc.

AGRADECIMENTOS

Agradecer sempre é difícil e não quero ser injusto esquecendo alguém, porém sinto-me na obrigação de fazê-lo e correr este risco.

Agradecer a Deus, que me deu a capacidade para ensinar e aprender.

À editora Casa do Código, que acreditou na ideia inicial do livro, e a toda sua equipe, que se comprometeu ajudando incansavelmente para termos um trabalho de qualidade.

À minha família, que me apoiou, especialmente a minha esposa que entendeu as muitas horas de ausência.

Aos meus colegas, que dirigiram o carro no trajeto casa/trabalho e trabalho/casa pela colaboração e aos momentos de descontração que eles proporcionaram.

À cooperativa Sicoob MaxiCrédito, na pessoa do Carlos Bittarello, que acreditou no Ember (pelo menos, na maior parte do tempo) como sendo uma alternativa eficaz para desenvolver aplicações Web e com certeza não se arrependeu.

SOBRE O AUTOR

Desenvolvedor de software desde 2008, Clairton Rodrigo trabalhou em sistemas Web em várias linguagens, como PHP, Java e Ruby no server; e HTML, CSS e JavaScript no front.

Graduado em Sistemas de Informação e especialista em Engenharia de Software e Programação, também tem interesse em Padrões de Projetos, Orientação a Objetos, Orientação a Aspectos, Metaprogramação e Componentização, independente da tecnologia.

No momento, está trabalhando como desenvolvedor full-stack na Sicoob Maxicrédito (http://maxicredito.coop.br) e como professor na Horus Faculdades (http://horus.edu.br). Além disso, também contribui com alguns projetos que podem ser vistos em seu GitHub (*@clairton*) e compartilha coisas sobre desenvolvimento de software no Facebook (*@clairton.desenvolvedor*).

Conheceu o framework Ember, ainda na versão 1.X, e desde então tem se interessado ainda mais por esse pretensioso SPA.

SOBRE O LIVRO

As aplicações desenvolvidas ficam cada vez mais especialistas, e os frameworks e bibliotecas que usamos para construir os sistemas também seguem a mesma tendência. Tendência esta, que aponta para o desenvolvimento de uma API que roda na nuvem, com uma ou várias aplicações clientes, sejam elas Web ou nativa. Isolando as responsabilidades de cada uma dessas partes, back-end e front-end, temos inúmeras vantagens.

Com o advento da era Web 2.0, com as especificações do HTML5, CSS3 e API do JavaScript, além do surgimento de ferramentas como Node, Bower e Npm, a interface na qual o usuário interage também tem a tendência de não ser gerada com Java, Ruby, PHP ou até mesmo o Node.js, mas sim se autogerenciar e tomar as decisões de como se comportar mediante apenas aos dados que recebe em JSON.

Com conceitos avançados, antigamente presentes apenas em aplicações desktop - como binding de propriedades entre o JavaScript e HTML, componentização e gerenciamento de dependências -, o Ember.js aparece como uma ótima opção. Sem ter uma grande empresa o apoiando, mas com uma comunidade vibrante, o Ember recebe novidades a passos largos.

Com ele, é possível reutilizar bibliotecas já escritas e amplamente conhecidas na comunidade (como o Twitter Bootstrap e o Moment.js), persistir dados utilizando requisições assíncronas de forma transparente, escrever testes unitários, de integração ou de aceitação de maneira organizada. O Ember

também possibilita escrever componentes e distribuí-los para outras aplicações em forma de *add-on*.

Seu código poderá usar os recursos mais atuais da ECMAScript, delegando a preocupação da incompatibilidade para um transpilador que também minificará o seu código para ambiente de produção.

Apesar de não tratar de assuntos avançados, para um bom aproveitamento do livro, seria interessante o leitor ter uma base das tecnologias CSS, HTML e JavaScript.

Agora, convido-o a conhecer e se aprofundar nesse framework. Você nunca mais será o mesmo.

Sumário

1 Introdução ao Ember 1

2 Iniciando uma aplicação 8

3 Ember-cli: instalação e arquitetura 14

4 Objetos Ember 20

5 Componentes do Ember 24

6 Roteamento 29

7 A engine de templates Handlebars 38

8 Ações e eventos 44

9 Os modelos da aplicação 54

10 Configure o adaptador 61

11 Relacionando pessoas com telefones 69

12 Personalizando o JSON 75

13 Criando um tipo de atributos - Transformadores 81

14 Criando um helper para exibir a data por extenso 85

15 Disponibilizando meses do ano por extenso através de um serviço 89

16 Adicionando uma nova entrada na agenda 93

17 Definindo um idioma na inicialização da agenda 102

18 Reutilizando a lógica de criar para editar uma pessoa 109

19 Testando a criação de uma nova pessoa 117

20 Tratando os erros de validação definidos no servidor 128

21 Adicionando Twitter Bootstrap como dependência 137

22 O processo de construção 154

23 Add-ons no Ember 160

24 E agora, quem poderá nos defender? 168

Capítulo 1

INTRODUÇÃO AO EMBER

No decorrer dos anos, criamos várias tecnologias, e cada uma dessas tecnologias tinha seu objetivo bem definido. Criamos o SQL para armazenar nossos dados de uma maneira organizada e padronizada, para que depois fosse possível recuperá-los.

Criamos várias linguagens de programação para implementar as regras de negócio das aplicações integrando essas linguagens com as implementações do SQL. E quando estamos em uma aplicação Web, usamos o HTML para mostrar as informações e, com o CSS, melhoramos a aparência. Tradicionalmente, usamos o poderoso JavaScript apenas para dar interatividade.

A era do caos iniciou com todas essas tecnologias em um único arquivo:

```
<?php include_once 'connection.php';?>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="<?=$_SESSION['PATH']?>/vendor.cs
s">
  </head>
  <body>
      <table>
          <tr>
              <th>Id</th>
              <th>Name</th>
```

```
          </tr>
<?php
            if(!$connection){
                echo '<script type="text/javascript">alert("Erro
ao conectar com Banco de Dados");</script>';
            }else{
                $sql = $connection->prepare('SELECT id, name FROM
 clients ORDER BY name');
                $sql->execute();
                $count = 0;
                if ($sql->columnCount())  {
                while ($row = $sql->fetch(PDO::FETCH_ASSOC)) {
                    $count++;
                  echo "<tr><td>$row['id']</td><td>$row['name']</
td></tr>";
                }
                }else {
                  echo '<tr colspan="2" style="color: red;">Não f
oram encontrados registros</tr>';
                }
            }
?>
      </table>
    <script type="text/javascript">
        alert('Registros encontrados <?=$count?>');
    </script>
  </body>
</html>
```

Como você pode ver em nosso arquivo, sugestivamente chamado de `espaguete.php`, temos as mais diversas más práticas, que dificultam a correção de erros e adição de novas funcionalidades. Uma delas é que as variáveis são criadas em um arquivo chamado `connection.php`, adicionado ao atual contexto, não tendo controle de escopo.

Outro fator que agrava a manutenibilidade de uma aplicação é que temos o PHP recuperando dados de um banco de dados, e depois escrevendo HTML e JavaScript. Perceba que todas as tecnologias estão fortemente acopladas.

A linguagem PHP, usada aqui, é meramente ilustrativa e poderíamos estar usando com certeza um arquivo `.jsp` como scriplets, ou ainda um `.asp` ou `.erb`. Entretanto, o foco não é a linguagem, mas a forma como as tecnologias eram mescladas.

Depois de um longo tempo, o padrão de projetos MVC (*Model-View-Controller*) aparece para organizar essa bagunça, delegando responsabilidades. De certa forma, utilizando essa técnica, nossos scripts SQL não precisariam ser misturados com o HTML, pois temos o local indicado para colocá-los, ou seja, no Model; enquanto que o HTML também tem o seu lugar ideal, na View, que é responsável pela interação com o usuário.

Mas ainda assim, para o HTML ser gerado dinamicamente, precisamos dos recursos de uma linguagem de programação, pois ele é apenas uma linguagem de marcação. Podemos usar como exemplo a criação de uma tabela, com os dados encontrados no banco de dados.

Outro acontecimento que melhorou essa condição foi a descoberta do AJAX (*Asynchronous JavaScript and XML*). Enfatizo aqui a descoberta, pois sua criação remonta ainda na época que JavaScript não era chamado de JavaScript. O AJAX é uma forma de buscar novos conteúdos no servidor para mostrar o navegador sem necessariamente recarregar a página inteira. Esses dados advindos do servidor geralmente apresentavam-se no formato XML, que era parseado pelo JavaScript. Este também manipulava o DOM adicionando novos elementos à página e tomando decisões conforme os valores obtidos.

Os endereços que as requisições AJAX visitavam para recuperar o XML não tinham um padrão de URL ou verbos, e era

comum ter um mesmo endereço para várias operações utilizando um parâmetro que identificava a operação. Por exemplo, havia o endereço `/pessoa`, e quando era para salvar uma pessoa, o parâmetro `xpto` tinha o valor de `S`, resultando no endereço `/pessoa?xpto=S`; quando era para remover o parâmetro, deveria ser `D`, resultando no endereço `/pessoa?xpto=D`. Em outros casos mais difíceis, o próprio recurso a ser acessado tinha um código, dificultando ainda mais a leitura, como `?recurso=123&xpto=D`.

Como podem ver, não se tinha um consenso de padrão entre as empresas e desenvolvedores, e cada um acabava por criar seu próprio método de escrita. Havia muita confusão para saber qual endereço eu poderia chamar para remover um registro. Poderia fazer uma requisição HTTP com o verbo `GET` para `pessoa/1`, ou até mesmo ser `pessoa/1?action=remover`, ou ainda um `?recurso=123&xpto=D`.

Foi então que, em uma sacada de mestre, alguém acorda e redescobre os outros métodos existentes no HTTP, além do `GET` e do `POST`. Pensando em uma maneira inteligente de usá-los, nomeou esse conceito de RESTful, aplicando aos verbos HTTP `PUT` e `DELETE` para atualizar e remover, respectivamente, um recurso do servidor; o `POST` para criar; e o `GET` para recuperar.

Assim, ao sabermos que o recurso que estamos acessando se chama `pessoa`, já temos desenhado quais os possíveis endereços para interagir:

Operação	Verbo HTTP	Endereço
Criar Pessoa	POST	/pessoa
Atualizar Pessoa com o id 1	PUT	/pessoa/1

Remover a Pessoa com id 1	DELETE	/pessoa/1
Recuperar Pessoa com id 1	GET	/pessoa/1
Recuperar todas as Pessoas	GET	/pessoa

Posteriormente, o conceito de colocar hyperlinks em cada resposta também ajudou a dar maior interatividade, denominado HATEOAS. Ele consiste em colocar endereços indicando as possíveis direções que podem ser tomadas em cada uma das requisições AJAX.

As tecnologias usadas para desenvolver aplicações web contam com gerenciamento de dependências e de ambientes. Porém, ainda não existia algo para gerenciar especificamente as dependências da View do MVC. Isso era feito copiando arquivos e colocando-os dentro do projeto, ou apenas adicionando um `<script src="" />` apontando para o endereço dele.

Em determinado momento, alguém teve a ideia de rodar JavaScript no back-end e chamá-lo de Node.js. Com isso, o JavaScript ganhou o que era necessário para construir um ecossistema independente, tornando possível criar módulos JavaScript, definindo de quais outros módulos ele depende.

Por exemplo, temos um módulo chamado `foo`, que depende do módulo `bar` na versão 0.1.0. E definimos isso no arquivo `package.json` do projeto:

```
//package.json

{
  "name": "foo",
  "dependencies":{
      "bar": "0.1.0"
```

```
  }
}
```

Agora o cenário está pronto, e a interface web pode gerenciar ela mesma. O movimento das aplicações de uma única página ganha força, dando origem ao padrão de projetos conhecido como SPA (*Single Page Application*). Vários frameworks surgem nessa arquitetura, entre eles o Ember, tema deste livro.

A história do Ember remonta ao projeto Sproutcore, protagonizado inicialmente pela Apple, e atualmente pela comunidade. É dele que, em 2011, Yehuda Katz iniciou um fork denominado *EmberJS*. Um tempo depois também chegou o *Ember-data*. Os primeiros commits do repositório são responsáveis pela persistência de dados, encapsulando a comunicação com sistemas remotos ou até mesmo a API localstorage do HTML5.

Mais algum tempo depois, surge o *ember-cli*, padronizando a resolução de problemas de gerenciamento de dependências, minimização de assets, usando a mais nova especificação ECMAScript e transpilando para a versão usada nos browsers.

Assim como os outros SPA que usam JSON, o Ember não depende de uma tecnologia específica para o servidor, ele apenas precisa ler e escrever JSON. Conforme a tecnologia, você pode encontrar bibliotecas que exigem pouco ou nenhum esforço de integração.

Com o Ember, podemos desenvolver qualquer aplicação web, mas querer carregar todos os arquivos antes de funcionar pode não ser indicado para aplicações que necessitam ter um início muito rápido como requisito.

Além das vantagens do *two way data-biding*, gerenciamento de dependências e assets, o Ember, através do `ember-cli` , resolve a criação de código padrão. Ele gera uma boa parte de instruções repetitivas, em uma estrutura definida, desacoplando as classes e facilitando os testes.

Apesar de não ter o patrocínio de uma grande empresa, como no caso da relação entre o Angular.js e a Google, sua comunidade se expande seguramente e conta com cases de sucesso no Yahoo!, Zendesk, Discourse e Groupon.

O Ember é um MVC, assim como o Ruby on Rails, VRaptor, Zend etc. Mas também é um SPA, o que muda totalmente a abordagem, em uma aplicação tradicional. Com algumas exceções, todo o HTML, CSS e JavaScript é gerado no servidor e o Browser apenas o interpreta; enquanto no SPA, depois de ser baixado do servidor na primeira requisição, toda a comunicação a seguir é feita apenas com requisições assíncronas, e a própria interface passa a se gerenciar.

Neste livro, abordaremos os conceitos do Ember, enquanto construímos uma agenda telefônica.

Capítulo 2

INICIANDO UMA APLICAÇÃO

No decorrer deste livro, desenvolveremos uma aplicação que chamaremos de agenda, será uma aplicação simples que vai interagir com uma API já existente, publicada no endereço `https://agenda.clairton.eti.br`. Essa agenda terá basicamente o nome da pessoa e seus números de telefones.

Podemos iniciar uma aplicação Ember ao adicionar ele e suas dependências em um projeto através de uma tag `script`, apontando o atributo `src` para o caminho do arquivo. Então, vamos fazer um primeiro teste. Criaremos um HTML básico, adicionando três dependências necessárias além do Ember. Vamos precisar do jQuery e do Ember Template Compiler também.

```
<!DOCTYPE html>
<html>
<head>
  <title>Agenda</title>
  <meta charset="utf-8">
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1
/jquery.min.js"></script>
  <script src="http://builds.emberjs.com/release/ember-template-c
ompiler.js"></script>
  <script src="http://builds.emberjs.com/release/ember.js"></scri
pt>
```

```
</head>
<body>
</body>
</html>
```

Assim já podemos instanciar o Ember dentro de uma tag `script`:

```
<script>
  var App = Ember.Application.create();
</script>
```

Para que algo seja exibido, também é necessário criarmos um template, que por sua vez ficará dentro de uma tag `script` com o type `text/x-handlebars`.

```
<script type="text/x-handlebars">
  <h1>Agenda</h1>
</script>
```

Nossa página HTML ficaria com o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
<head>
  <title>Agenda</title>
  <meta charset="utf-8">
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1
/jquery.min.js"></script>
  <script src="http://builds.emberjs.com/release/ember-template-c
ompiler.js"></script>
  <script src="http://builds.emberjs.com/release/ember.js"></scri
pt>
  <script>
    var App = Ember.Application.create();
  </script>
</head>
<body>
  <script type="text/x-handlebars">
    <h1>Agenda</h1>
```

```
  </script>
</body>
</html>
```

Se salvarmos esse HTML estático em um arquivo e abrirmos com um navegador, temos a seguinte exibição:

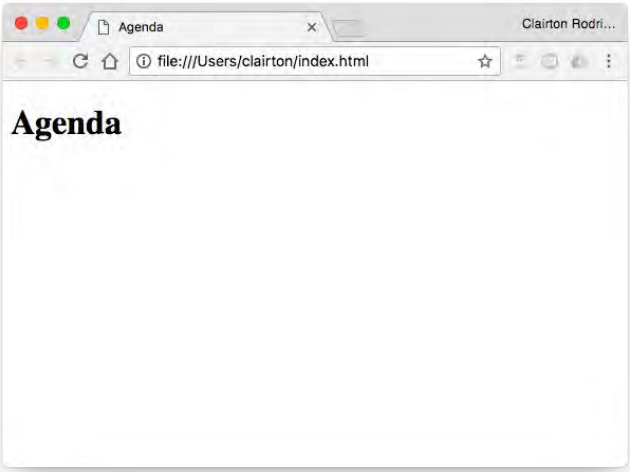

Figura 2.1: Exibição do HTML estático

Para ter um conteúdo dinâmico, vamos criar um controlador, a parte **C** do MVC, que serve como ligação ente o **M** de modelo e o **V** de visão. Esse controlador que chamaremos de `lista` terá uma propriedade denominada `nome` e herdará da implementação de `Ember.Controller`:

```
<script>
  var App = Ember.Application.create();
  App.ListaController = Ember.Controller.extend({
    nome: 'Clairton'
```

```
  });
</script>
```

Para exibir essa propriedade, o Ember procurará uma visão com o mesmo nome. A parte da visão no Ember é composta pelas Rotas e Templates. O que criaremos agora é um template com o atributo `data-template-name` de `lista`. Dentro desse template, usaremos o HTML normalmente, além de termos a marcação `{{` que serve para fazer o bind de propriedades, no caso da propriedade chamada `nome`.

```
<script type="text/x-handlebars" data-template-name="lista">
  {{nome}}
</script>
```

Se atualizarmos o navegador onde tínhamos aberto o arquivo, depois dessas alterações salvas, não haverá nenhuma modificação ainda, pois precisamos criar a Rota para que essa visão seja exibida. Essa Rota será mapeada para o endereço `/`. O mapeamento das rotas é feito através de uma função passada para `App.Router.map`:

```
<script>
  var App = Ember.Application.create();
  App.ListaController = Ember.Controller.extend({
    nome: 'Clairton'
  });
  App.Router.map(function() {
    this.route('lista', {path: '/'});
  });
</script>
```

Também é preciso adicionar a marcação `{{outlet}}` ao template principal, para que ali seja renderizado o template `lista`:

```
<script type="text/x-handlebars">
  <h1>Agenda</h1>
  {{outlet}}
</script>
```

O trecho inteiro de código será:

```
<!DOCTYPE html>
<html>
<head>
  <title>Agenda</title>
  <meta charset="utf-8">
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1
/jquery.min.js"></script>
  <script src="http://builds.emberjs.com/release/ember-template-c
ompiler.js"></script>
  <script src="http://builds.emberjs.com/release/ember.js"></scri
pt>
  <script>
    var App = Ember.Application.create();
    App.ListaController = Ember.Controller.extend({
      nome: 'Clairton'
    });
    App.Router.map(function() {
      this.route('lista', {path: '/'});
    });
  </script>
</head>
<body>
  <script type="text/x-handlebars">
    <h1>Agenda</h1>
    {{outlet}}
  </script>
  <script type="text/x-handlebars" data-template-name="lista">
    {{nome}}
  </script>
</body>
</html>
```

Salvando esse conteúdo em um arquivo e abrindo-o em qualquer navegador, temos:

Figura 2.2: Exemplo básico

Para pôr o atributo `nome` em um input, utilizamos o *helper* `input`, setando o atributo `value` com o nome.

```
<script type="text/x-handlebars" data-template-name="lista">
  {{input value=nome}}
</script>
```

Assim, toda vez que o atributo `nome` for alterado, o valor no JavaScript também será, e vice-versa.

Até agora, essa forma de desenvolvimento funcionou, mas nosso código está sendo colocado em apenas um arquivo, e isso não dará certo em aplicações grandes. Por isso, no próximo capítulo, vamos instalar o `ember-cli`, uma ferramenta que ajuda no desenvolvimento com Ember, com o gerenciamento de dependências, geração de código repetitivo, minimização e transpilação de código, e muito mais.

CAPÍTULO 3

EMBER-CLI: INSTALAÇÃO E ARQUITETURA

Com o uso do `ember-cli` (*Ember Command Line Interface*), podemos definir o Ember como uma dependência, e não em uma tag `script` . Ele, por usa vez, é instalado como um módulo do Node.js.

Não abordaremos a instalação do Node.js, mas você pode encontrar mais informações no site do projeto em http://nodejs.org.

Tendo o Node.js instalado, o `ember-cli` pode ser adicionado ao seu sistema operacional com um comando executado no terminal. Se você está em um ambiente Linux, possivelmente será necessário estar com poderes de superusuário.

```
npm install -g ember-cli
```

Após a instalação, terá disponível no terminal do seu sistema operacional o comando `ember` . Vamos criar um projeto e transferir a estrutura do que tínhamos feito anteriormente para a estrutura do `ember-cli` .

```
ember new agenda
```

Esse comando gerará uma nova aplicação com diretórios muito parecidos com uma aplicação MVC tradicional. Sendo ainda mais específico, podemos dizer com as mesmas características do Ruby on Rails.

Entrando na pasta `agenda`:

```
cd agenda
```

E usando o comando:

```
tree -d -L 2 -I 'node_modules|bower_components'
```

Ignoramos as pastas `node_modules` e `bower_components`, porque são pastas de arquivos temporários, assim como a `tmp` e a `dist`. Então, temos:

```
├── app #arquivos da aplicação
│   ├── components #componentes
│   ├── controllers #controllers
│   ├── helpers #helpers
│   ├── models #modelos
│   ├── routes #rotas
│   ├── styles #arquivos de estilos
│   └── templates #templates do handlebars
├── config #arquivos de configuração
├── public #assets publicos
├── tests #testes da aplicação
│   ├── helpers #helpers para testes
│   ├── integration #testes de integração
│   └── unit #testes unitários
└── vendor #arquivos de terceiros
```

O `ember new`, logo após criar os diretórios e arquivos, rodará o comando `npm install` que baixará os módulos do Node.js, que estão declarados no arquivo `package.json`. Ele também vai

rodar o comando `bower install` para baixar os componentes do Bower especificados no arquivo `bower.json`.

Executando o comando `ember server` dentro do diretório do projeto que criamos, temos:

```
$ ember server
Could not start watchman
Visit https://ember-cli.com/user-guide/#watchman for more info.
Livereload server on http://localhost:49153
Serving on http://localhost:4200/

Build successful - 5627ms.



Slowest Nodes (totalTime => 5% )                | Total (avg)
------------------------------------------------+------------------
---
Babel (16)                                      | 4762ms (297 ms)
Concat (8)                                      | 288ms (36 ms)
```

E visitando o endereço `localhost:4200` no navegador:

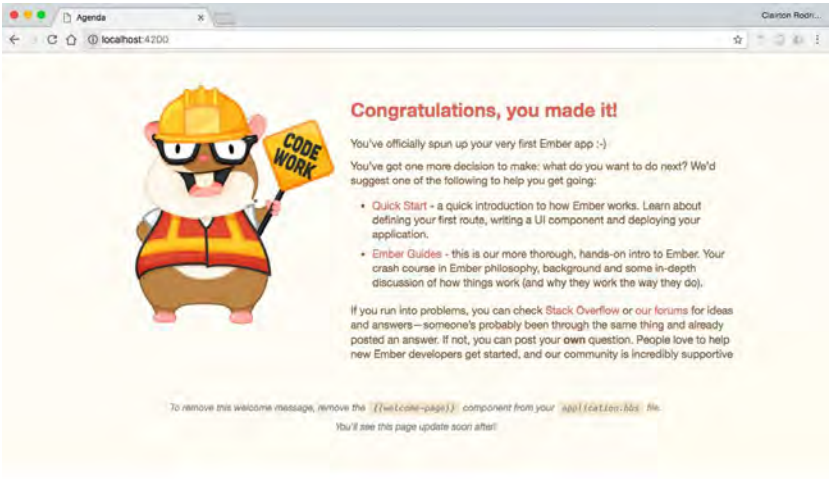
Agenda
localhost:4200
Congratulations, you made it!
You've officially spun up your very first Ember app :-)
You've got one more decision to make: what do you want to do next? We'd suggest one of the following to help you get going:
Quick Start - a quick introduction to how Ember works. Learn about defining your first route, writing a UI component and deploying your application.
Ember Guides - this is our more thorough, hands-on intro to Ember. Your crash course in Ember philosophy, background and some in-depth discussion of how things work (and why they work the way they do).
If you run into problems, you can check Stack Overflow or our forums for ideas and answers—someone's probably been through the same thing and already posted an answer. If not, you can post your own question. People love to help new Ember developers get started, and our community is incredibly supportive
To remove this welcome message, remove the {{welcome-page}} component from your application.hbs file.
You'll see this page update soon after!

Figura 3.1: Tela inicial

Com o `ember-cli` , cada código tem seu lugar e podemos usar os geradores de código. Como primeiro passo a partir de agora, seria bom criarmos o controlador de `lista` :

```
ember generate controller lista
```

Editamos o arquivo `app/controllers/lista.js` e deixamos com a implementação que tínhamos no controlador `lista` do capítulo anterior:

```
//app/controllers/lista.js

export default Ember. Controller.extend({

  nome: 'Clairton'

});
```

Agora vamos gerar a rota `lista`:

```
ember generate route lista
```

Além de criar o arquivo `app/templates/lista.hbs`, o comando também criará o `app/routes/lista.js` e adicionará uma configuração apontando para eles em `app/routes.js`. Lá passaremos como parâmetro o atributo `path`, indicando que a rota fique publicada na raiz.

```
//app/router.js
...
this.route('lista', {path: '/'});
...
```

Também editaremos o arquivo de template `app/templates/lista.hbs`, colocando o conteúdo do template `lista` do capítulo anterior.

```
{{!app/templates/lista.hbs}}

{{input value=nome}}
```

Agora para ver a página que criamos, é necessário deixar o arquivo do template principal `app/templates/application.hbs` assim:

```
{{!app/templates/application.hbs}}

<h1>Agenda</h1>

{{outlet}}
```

Visitando o navegador no endereço `http://localhost:4200` temos:

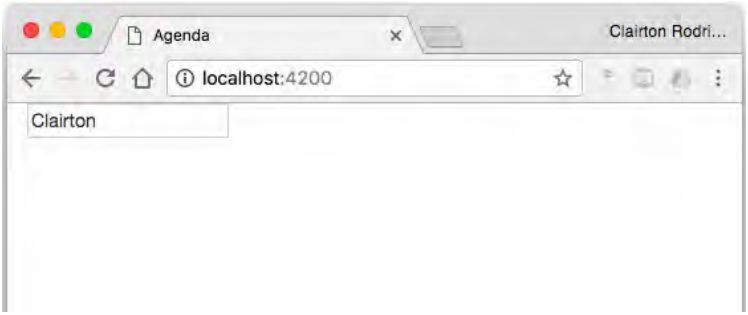

Figura 3.2: Primeiro componente

Lógico que não temos nada de diferente ainda. Antes de prosseguirmos, aprenderemos uma pouco sobre os objetos Ember, e é isso que abordaremos ao virar a página.

Capítulo 4

OBJETOS EMBER

Muito além de simples objetos do JavaScript, uma instância de um objeto Ember acrescenta várias facilidades e funcionalidades que os objetos JavaScript nativos não suportam, como a observação de alterações de valor de propriedades. O Ember também fornece um sistema de classes, além de recursos como mixins e métodos construtores.

Podemos usar recursos da ECMAScript 6, como interpolação de strings, arrow functions e constantes, que depois são transpilados para a versão atual suportada pelos navegadores. Nesta parte do livro, veremos alguns trechos de código, e posteriormente voltaremos a desenvolver nossa agenda.

Para criarmos um objeto Ember, invocamos o método construtor da classe `Object` , de onde todas as classes do Ember descendem, passando como parâmetro um array de chave/valor, em que a chave será o nome da propriedade:

```
let pessoa = Ember.Object.create({
 nome: 'Clairton' ,
 sobrenome: 'Heinzen'
});
```

Para recuperar o valor de uma propriedade, usamos o método

`get`, passando o nome do atributo como parâmetro:

```
console.log(pessoa.get('nome'));
```

Já para alterar o valor do atributo, passamos também o novo valor invocando o método `set`:

```
pessoa.set('nome', 'Pedro');
```

Perceba que podemos criar novos atributos que são resultados do valor de outros atributos. No nosso caso, `nome` e `sobrenome` compõem o nome completo, mas para isso precisamos criar uma classe nova, herdando de `Ember.Object`.

Para criar um atributo computado, usamos o método `Ember.computed`, que recebe como parâmetro o nome do atributo que forma o novo atributo (podendo ser mais de um nome de atributo), e por último uma `function` que retorna o valor desse novo atributo.

```
let Pessoa = Ember.Object.extend({
...
 nomeCompleto: Ember.computed('nome', 'sobrenome', function(){
  return `${this.get('nome')} ${this.get('sobrenome')}`;
 })
});
```

Depois, podemos instanciar a classe `Pessoa` que criamos:

```
let pessoa = Pessoa.create({
  nome: 'João',
  sobrenome: 'Silva'
});
```

Podemos também observar a alteração de valores dos atributos,

usando o método `Ember.observer` . Ele recebe como parâmetros o(s) nome(s) do(s) atributo(s), e por último uma `function` .

No exemplo a seguir, temos um observador que chamamos de `aoAlterarNome` . Toda vez que o atributo `nomeCompleto` for alterado, ele exibirá uma mensagem do console.

```
let Pessoa = Ember.Object.extend({
...
 aoAlterarNome : Ember.observer('nomeCompleto', function(){
   console.log(`nome alterado para ${this.get('nomeCompleto')}`);
 });
});
```

O Ember também oferece facilidades para trabalhar com coleções. Vamos supor que temos uma coleção de telefones em `pessoa` , e em cada telefone temos duas propriedades que são `numero` e `prefixo` . Agora, necessitamos saber quais os prefixos presentes. Criamos a propriedade:

```
let Pessoa = Ember.Object.extend({
 telefones: []
});
```

Adicionamos alguns itens na coleção:

```
let pessoa = Pessoa.create({nome: 'Clairton'});
let residencial = Ember.Object.create({
 prefixo: 49,
 numero: '33333333'
});
pessoa.get('telefones').pushObject(residencial);
let comercial = Ember.Object.create({
 prefixo: 47,
 valor: '99999999'
});
pessoa.get('telefones').pushObject(comercial);
```

Para recuperar os prefixos da coleção, se estivéssemos em JavaScript puro, teríamos algo como:

```
let prefixos = [];
for(var i = 0, j = pessoa.get('telefones.length'); i < j; i++){
  prefixos.push(pessoa.get('telefones')[i]);
}
console.log(prefixos);
```

Mas com o Ember, podemos usar o método `computed` passando como parâmetro o nome do atributo da lista de objetos que desejamos somar. E entre eles, usamos o `@each` que observará o atributo `prefixo` de cada item da coleção. Na implementação do método que nomeamos de `prefixos`, invocamos o método `mapBy`, passando também o atributo `prefixo`.

```
let Pessoa = Ember.Object.extend({
 ...
 prefixos: Ember.computed('telefones.@each.prefixo', function(){
  return this.get('telefones').mapBy('prefixo');
 })
});
```

Assim, a cada registro adicionado ou removido em `telefones`, a lista em `prefixos` será atualizada.

Existem outras funcionalidades úteis das coleções que o Ember denomina, como `Enumerable`. Entre elas, podemos filtrar itens, somar valores de atributos, achar o maior ou menor valor, retornar o primeiro ou o último objeto etc.

Capítulo 5

COMPONENTES DO EMBER

O mundo Web é feito de várias especificações conduzidas por um consórcio de empresas e uma lista denominado W3C (*World Wide Web Consortion*). São as especificações que fazem com que uma página web escrita anos atrás ainda possa ser vista nos navegadores em suas versões atuais.

Uma especificação que está ganhando força é a do Web Components, composta por 3 partes que procuram resolver problemas comuns encontrados no desenvolvimento Web:

- **Custom Elements:** criar elementos customizados para usar e redistribuir;
- **HTML Imports:** importar partes de HTML para poder importar os Custom Elements;
- **Shadow DOM:** isolar contextos de CSS, assim um determinado CSS pode apenas ser aplicado a um elemento HTML e seu conteúdo.

Enquanto essas especificações ainda não estão prontas, o Ember disponibiliza uma forma de criar, testar e publicar componentes. A componentização promove manutenibilidade,

E-book gerado especialmente para Matheus Rodrigues F da Silva - matheusuniversal@yahoo.com.br

pois ajuda no reaproveitamento de código e isolamento de contexto, quebrando um sistema de grande complexidade em pequenas partes reutilizáveis.

Já vimos um pouco de componentes quando criamos um campo de texto e mostramos o conceito de bind. Nós temos um atributo `nome` no controlador `lista`:

```
//app/controllers/lista.js
export default Ember. Controller.extend({
  nome: 'Clairton'
});
```

E a chamada ao componente `input`:

```
{{!app/templates/lista.hbs}}

{{input value=nome}}
```

O `input` que usamos aqui é um componente, ao qual passamos o atributo `nome` para o parâmetro `value`. Os componentes do Ember ainda não implementam integralmente o conceito de Web Components. Mesmo que isso esteja em seus objetivos, o isolamento de contextos do CSS ainda não é possível.

Para criarmos um componente, utilizamos o comando:

```
ember generate component mostra-pessoa
```

Assim como quando criamos a rota `lista`, o `generate component` gerará 3 arquivos: um JavaScript, um template e um teste. Adicionaremos o nome da pessoa no arquivo de template.

```
//app/components/mostra-pessoa.js
```

```
import Ember from 'ember';

export default Ember.Component.extend({});
```

```
{{!app/components/mostra-pessoa.hbs}}

Nome: {{pessoa.nome}} - {{pessoa.sobrenome}}
```

Posteriormente, para usarmos esse nosso componente, passamos como parâmetro a pessoa que queremos mostrar:

```
{{mostra-pessoa pessoa=pessoa}}
```

Considerando que a variável `pessoa` tenha o atributo `nome` preenchido com `Clairton`, o resultado em HTML seria semelhante a este:

```
<div>
  Nome: Clairton
</div>
```

Como é possível perceber, a saída está dentro de uma tag `div`. Mas ela pode ser personalizada sobrescrevendo o atributo `tagName`.

```
//app/components/mostra-pessoa.js

import Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'span'
});
```

Dessa forma, ao usar o componente:

```
{{mostra-pessoa pessoa=pessoa}}
```

A saída HTML seria algo como:

```
<span>
  Nome: Clairton
</span>
```

Podemos passar também um determinado conteúdo para que o componente renderize. Além dos parâmetros, é possível usar a marcação `{{yield}}`, por exemplo:

```
{{!app/components/mostra-pessoa.hbs}}

{{yield}}

Nome: {{pessoa.nome}} - {{pessoa.sobrenome}}
```

Depois, para usarmos esse nosso componente, além do parâmetro `pessoa`, passamos também o que queremos mostrar:

```
{{#mostra-pessoa pessoa=pessoa}}
  <div>Esse conteúdo renderizará dentro do componente onde está a
 marcação {{{yield}}}</div>
{{/mostra-pessoa}}
```

Dessa forma, teremos o seguinte resultado:

```
<div>
  <div>Esse conteúdo renderizará dentro do componente onde está a
 marcação {{yield}}</div>
  Nome: Clairton
</div>
```

Na invocação do componente `mostra-pessoa`, também usamos `{{{` (três chaves), que servem para escapar a anotação de duas chaves `{{` do Handlebars.

Podemos também usar o atributo `classNameBindings` para

customizar as classes CSS. Para exemplificar, poderíamos alterar a classe se o nome tivesse apenas uma letra, pois provavelmente ele não seria válido.

```
//app/components/mostra-pessoa.js

import Ember from 'ember';

export default Ember.Component.extend({
  classNameBindings: ['umaLetra::nome-errado'],

  umaLetra: Ember.computed('pessoa.nome', function(){
    return this.get('pessoa.nome') <= 1;
  }),

  pessoa: Ember.Object.create({
    nome: 'J'
  });
});
```

Terá uma saída semelhante com essa:

```
<div class="nome-errado">
  Nome: J
</div>
```

Os componentes, se desenvolvidos com qualidade, são altamente testáveis. Podemos passar parâmetros com valor diversos e testar se o comportamento é o esperado. Ao longo dos próximos capítulos, veremos esse e mais alguns conceitos ligados ao Ember e como utilizá-los.

Capítulo 6

ROTEAMENTO

As diferentes funcionalidades em uma aplicação SPA, ou até mesmo uma RESTful, estão em endereços diferentes para cada recurso e operação. Por exemplo, em nossa aplicação de agenda de contatos que estamos escrevendo, quanto necessitarmos ver os nomes das pessoas, você deverá ir ao endereço `/pessoas` , e se quiser apenas listar os telefones, logicamente irá para `/telefones` .

Poderíamos chamar isso de endereços inteligentes, pois eles falam por si só qual informação e operação referenciam. Dessa forma, sei que quando estou no endereço `pessoas/1` posso aferir que estou acessando os dados da `pessoa` com o identificador `1` .

Em uma aplicação Ember, temos o Router, que usa o mesmo conceito de um roteador de rede, que encaminha os pacotes ao seu destino. O Router fica no arquivo `app/router.js` e mapeia as configurações sobre as possíveis rotas da aplicação.

Ele recebe as requisições vindas pela URL e encaminha-as para a rota correta mapeada por ele. Podemos dizer que ele faz a tradução do endereço para o template correto, como um roteador de rede. Veja a seguir um exemplo de roteador:

```
//app/router.js

import Ember from 'ember';

export default Ember.Router.map(function() {
  /*
   * por padrão já existe uma rota application,
   * então não precisamos e não devemos criá-la novamente
   */
  //this.route('application');
});
```

Cada rota possui um template onde podemos mostrar os dados ao usuário. Para cada rota, temos também um controlador com o mesmo nome da rota, que guarda o estado da aplicação.

O uso dos controladores é desencorajado, sendo indicado usar o máximo possível de componentes, já que o `controller` tem a mesma função da parte JavaScript dos componentes. Segue um exemplo da rota principal de uma aplicação Ember, denominada `application`. No template dessa rota, encontraremos a anotação `{{outlet}}`, responsável por renderizar os conteúdos das rotas aninhadas a ela.

```
//app/routes/application.js

import Ember from 'ember';

export default Ember.Route.extend({
});
```

```
//app/controller/application.js

import Ember from 'ember';

export default Ember.Controller.extend({});
```

```
{{!app/templates/application.hbs}}
```

```
{{outlet}}
```

Como já comentado anteriormente, nossa aplicação agenda conterá nomes e telefones, então começaremos com uma lista de pessoas na rota `lista`. Para que a lista de pessoas possa ser recuperada, ela deve estar em um método chamado `model`.

```
//app/routes/lista.js

import Ember from 'ember';

export default Ember.Route.extend({
  pessoas: [{id: "1", nome: 'Clairton'}, {id:"2", nome: 'Rodrigo'
}],

  model(){
    return this.get('pessoas');
  }
});
```

Ainda temos de mostrar esses dados no template, e o faremos com o helper `each` para imprimir cada nome:

```
{{!app/templates/lista.hbs}}

{{input value=nome}}
<br/>

{{#each model as |pessoa|}}
  <label for="nome">Nome:</label>
  {{pessoa.nome}}<br/>
{{/each}}
```

Visitando o endereço `http://localhost:4200`, ficamos com uma tela parecida com essa:

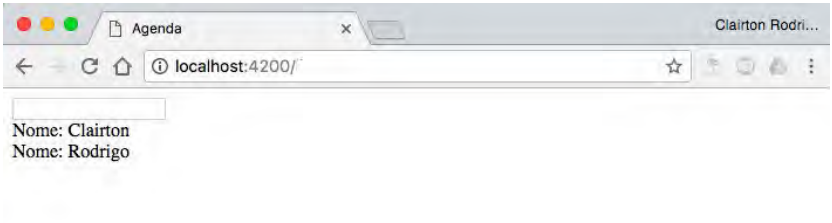

Figura 6.1: Listando Pessoas

Obviamente, quando nossa agenda tiver muitos contatos, precisaremos de uma forma de pesquisar as pessoas pelo nome. Com o Ember, é possível adicionar parâmetros de consulta (que aparecem no endereço). Como faremos customizações no controlador, atribuiremos `null` ao atributo `nome` e o adicionaremos ao atributo `queryParams` .

```
//app/controller/lista.js

import Ember from 'ember';

export default Ember.Controller.extend({
  nome: null,
  queryParams: ['nome'],
});
```

Precisamos também de algumas alterações na rota, na qual colocaremos o atributo `queryParams` dizendo que os registros devem ser recarregados a cada vez que o atributo `nome` tiver seu valor alterado com `refreshModel: true` :

```
//app/routes/lista.js

import Ember from 'ember';
```

```
export default Ember.Route.extend({
  pessoas: [{id: "1", nome: 'Clairton'}, {id:"2", nome: 'Rodrigo'
}],

  queryParams: {
    nome: {
      refreshModel: true
    }
  },

  model(params){
    return this.get('pessoas').filter((pessoa) => {
      return Ember.isEmpty(params.nome) || pessoa.nome.indexOf(pa
rams.nome) >= 0;
    });
  }
});
```

Também identificaremos o campo de entrada já existente, colocando uma mensagem `Filtrar por nome`, além de pôr as pessoas dentro de uma tabela.

```
{{!app/templates/lista.hbs}}

<label for="filtro-nome">Filtrar por nome</label>
{{input id="filtro-nome" value=nome}}
<br/>

<table>
  <thead>
      <tr>
        <th>Nome</th>
      </tr>
  </thead>
  <tbody>
      {{#each model as |pessoa|}}
        <tr>
          <td>
            {{pessoa.nome}}
          </td>
        </tr>
      {{/each}}
```

```
  </tbody>
</table>
```

Ao digitar no campo de pesquisa, pode-se perceber que o endereço da lista de pessoas será atualizado. Por exemplo, se preenchermos com `Clairton` :

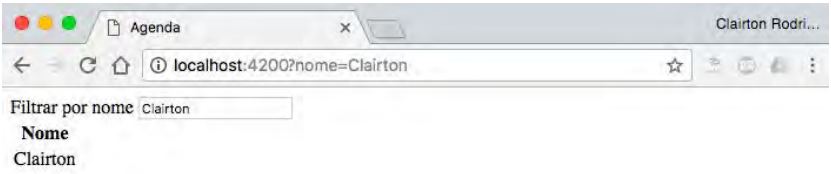

Figura 6.2: Filtrando Pessoas pelo nome

Usamos esse recurso para pesquisar a pessoa pelo nome, mas ele também pode ser usado para outras funcionalidades, como por exemplo paginar os resultados vindos do servidor.

Na rota, também temos um método com a seguinte assinatura `setupController(controler, model)` . Ele é uma espécie de callback que pode ser usado para setar alguma propriedade no controller ou no model.

Seguindo o princípio de convenção sobre configuração, o template usado tem o mesmo nome da rota, porém esse comportamento pode ser alterado sobrescrevendo o método `renderTemplate` :

```
...
renderTemplate() {
  this.render('outroNomeDeTemplateQueDesejo');
}
```

...

Depois de pesquisar pela pessoa em específico, precisamos ter uma forma de ver todos os dados dela (apesar de no momento termos apenas o nome). Para ter os dados dessa pessoa, criaremos uma rota aninhada que recebe como parâmetro o `id` da pessoa escolhida:

```
ember generate route pessoa
```

Precisaremos passar o id para essa rota, e faremos isso no router, na invocação do método `route`, ao passarmos um segundo parâmetro além do nome da rota. Esse segundo parâmetro é um hash, e usaremos a opção `path`:

```
//app/router.js

import Ember from 'ember';

export default Ember.Router.map(function() {

    this.route('lista', {path: '/'});
    this.route('pessoa', {path: 'pessoa/:id'});

});
```

Para dar a possibilidade de detalharmos a pessoa, enviando a aplicação para essa nova rota `pessoa`, colocaremos um `link` no nome dela. Para fazer isso, usaremos um helper do `Handlebars` chamado `link-to`, que recebe dois parâmetros: o nome da rota e o `id` da pessoa.

```
{{!app/templates/lista.hbs}}

<label for="filtro-nome">Filtrar por nome</label>
```

```
{{input id="filtro-nome" value=nome}}
<br/>

<table>
  <thead>
      <tr>
        <th>Nome</th>
      </tr>
  </thead>
  <tbody>
      {{#each model as |pessoa|}}
        <tr>
          <td>
            {{#link-to 'pessoa' pessoa.id}}{{pessoa.nome}}{{/link
-to}}
          </td>
        </tr>
      {{/each}}
  </tbody>
</table>
```

O helper `link-to` criará a saída `HTML` semelhante a essa:

```
<a href="pessoas/1">Clairton</a>
```

Quando clicarmos, ele direcionará para o endereço conforme o `id` da pessoa. Se a pessoa tiver o `id 1`, ele vai direcionar para `http://localhost:4200/pessoa/1`. Agora vamos usar o componente `mostra-pessoa` que criamos anteriormente no arquivo `app/templates/pessoa.hbs`:

```
{{mostra-pessoa pessoa=model}}
```

Precisamos também implementar o método `model` na rota `app/routes/pessoa.js` para retornar a pessoa:

```
//app/routes/pessoa.js

import Ember from 'ember';
```

```
export default Ember.Route.extend({
    pessoas: [{id: "1", nome: 'Clairton'}, {id:"2", nome: 'Rodrig
o'}],

    model(params){
        return this.get('pessoas').findBy('id', params.id);
    }
});
```

Com isso, o conteúdo do template `app/templates/pessoa.hbs` começa a ficar dessa forma:

Figura 6.3: Exibindo os dados de uma Pessoa

No próximo capítulo, vamos nos aprofundar especificamente no Handlebars, a engine de templates do Ember.

CAPÍTULO 7

A ENGINE DE TEMPLATES HANDLEBARS

O Ember utiliza a engine de templates Handlebars. Simples e poderosa, ela tem vários recursos que facilitam escrevermos nossa `view`. O Handlebars é uma extensão da *Linguagem de Template Mustache* e segue praticamente as mesmas regras, tanto que os templates escritos com a sintaxe do Mustache podem ser usados pelo Handlebars, por isso vamos nos referir a eles somente como Handlebars.

Os templates Handlebars consistem em usar marcações embutidas no HTML. Ele vem para suprir o que temos em linguagens back-end, como os arquivos `.erb` do Ruby, o `.jsp` do Java ou o `.asp` da plataforma Microsoft. Ele é uma forma de adicionarmos conteúdo dinâmico ao HTML.

No exemplo seguinte, temos uma tag `nome`, e sabemos disso porque tags estão entre `{{` e `}}`:

```
<p>Olá, meu nome é {{nome}}.</p>
```

E temos o objeto JavaScript com o conteúdo para a tag `nome`:

```
{nome: 'clairton'}
```

O Handlebars já vem instalado por padrão em um projeto com o `ember-cli`. Porém, para focarmos os estudos somente nele neste capítulo, vamos usá-lo através da tag `script`:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <script src="http://builds.handlebarsjs.com.s3.amazonaws.com/
handlebars.min-latest.js"></script>
</head>
<body>
</body>
</html>
```

Dentro dessa página, em uma nova tag `script`, vamos fazer o exemplo no qual o Handlebars vai substituir a tag `nome` pelo conteúdo `clairton`.

```
...
var fonte = "<p>Olha, meu nome é {{nome}}.</p>";
var template = Handlebars.compile(fonte);
var dados = { "nome": "clairton"};
var resultado = template(dados);
console.log(resultado);
...
```

No console, abrindo a ferramenta de desenvolvedores do seu navegador, o resultado teria o seguinte conteúdo:

```
<p>Olá, meu nome é clairton.</p>
```

Esse resultado pode ser obtido ao usarmos a manipulação do DOM com JavaScript puro. Mas o Handlebars nos proporciona algo mais legível e organizado.

Podemos criar *helpers*, que são tags que nos ajudam a

solucionar problemas, como capitalizar o nome. Para isso, criaremos um helper chamado `capitalize`. É indicado que um helper de palavras compostas seja separado por hífen (`-`).

```
Handlebars.registerHelper('capitalize', function(string) {
  return `${string.charAt(0).toUpperCase()}${string.slice(1).toLo
werCase()}`;
});
```

Assim, ao usarmos `{{capitalize nome}}` em nosso template, a saída do nome ficaria com a primeira letra maiúscula e o restante em minúscula.

```
...
```

```
Handlebars.registerHelper('capitalize', function(string) {
  return `${string.charAt(0).toUpperCase()}${string.slice(1).toLo
werCase()}`;
});
```

```
var fonte = "<p>Olha, meu nome é {{capitalize nome}}.</p>";
var template = Handlebars.compile(fonte);
var dados = { "nome": "clairton"};
var resultado = template(dados);
console.log(resultado);
...
```

E como resultado:

```
<p>Olá, meu nome é Clairton.</p>
```

O Handlebars nos traz muitos helpers prontos, como o `if`. Caso o conteúdo de `nome` for `false`, `undefined`, `null`, `""`, `0` ou `[]`, poderíamos exibir `'Nome Vazio'`:

```
{{if nome nome 'Nome Vazio'}}
```

Podemos chamar helpers dentro de helpers, entretanto, nesse caso precisamos circular a expressão com parênteses:

```
{{if (algum-helper-qualquer nome) nome 'Nome Vazio'}}
```

O helper `if` também pode ser usado em blocos. Nesse caso, a tag `{{#` inicia o bloco e tag `{{/` termina:

```
{{#if nome}}
 {{nome}}
{{/if}}
```

Mas o helper `if` também tem um `else`:

```
{{#if nome}}
 {{nome}}
{{else}}
  Nome Vazio
{{/if}}
```

O helper `if` testa apenas se um caso for positivo. Se você desejar executar algo para um teste falso, use o `unless`, que testa o inverso do helper `if`:

```
{{#unless nome}}
  Nome Vazio
{{else}}
 {{nome}}
{{/unless}}
```

Iterar sobre uma coleção de objetos é um recurso bem comum em linguagens de programação, e o Handlebars também tem o helper `each` para esse propósito. Este recebe como parâmetro uma lista de objetos, e também precisamos nomear a variável que

conterá as informações a cada iteração (no caso, chamamos de `pessoa`):

```
<ul>
  {{#each pessoas as |pessoa|}}
   <li>{{pessoa.nome}}</li>
  {{/each}}
</ul>
```

Podemos recuperar também o índice do objeto:

```
<ul>
  {{#each pessoas as |pessoa indice|}}
   <li>{{indice}} - {{pessoa.nome}}</li>
  {{/each}}
</ul>
```

Temos uma lista de pessoas em uma variável JavaScript e criaremos uma lista HTML.

```
var fonte = "<ul>{{#each pessoas as |pessoa|}} <li>{{pessoa.nome}
}</li> {{/each}}</ul>";
var template = Handlebars.compile(fonte);
var dados = [{"nome": "Clairton"}, {"nome": "Rodrigo"}];
var resultado = template(dados);
```

E a saída será:

```
<ul>
 <li>Clairton</li>
 <li>Rodrigo</li>
</ul>
```

O Handlebars também conta com tags para comentários. Na verdade, existem dois tipos de comentários:

```
{{!algum comentário}}
```

Este terá a saída em HTML:

```
<!--algum comentário-->
```

E outro que não aparecerá no HTML:

```
{{!--isso não gerará saída em html--}}
```

Além de serem poderosos, o Handlebars e o Ember podem ser estendidos ao adicionarmos novos recursos. No próximo capítulo, abordaremos ações e eventos.

Capítulo 8

AÇÕES E EVENTOS

O click em um botão, o pressionar de uma tecla e o movimento do mouse são apenas algumas das formas como interagimos com uma aplicação. Neste capítulo, você vai conhecer a simplicidade de trabalhar com ações no Ember, da rota para o controlador, do template para o controlador, do componente para o componente, e assim por diante.

Temos em nosso template `app/template/lista.hbs` a lista de pessoas de nossa agenda:

```
{{!app/templates/lista.hbs}}

<label for="filtro-nome">Filtrar por nome</label>
{{input id="filtro-nome" value=nome}}
<br/>

<table>
  <thead>
      <tr>
        <th>Nome</th>
      </tr>
  </thead>
  <tbody>
      {{#each model as |pessoa|}}
        <tr>
          <td>
            {{#link-to 'pessoa' pessoa.id}}{{pessoa.nome}}{{/link
-to}}
```

```
          </td>
        </tr>
      {{/each}}
  </tbody>
</table>
```

Seria interessante termos nesse template um botão que, ao ser clicado, mostrasse ao usuário uma tela para inserir uma nova pessoa em sua agenda. São para essas necessidades que existem as ações.

Mas para ter a possibilidade de criar um registro novo para pessoa, precisamos também de uma nova rota, que criaremos executando o comando:

```
ember generate route nova
```

Editaremos o arquivo `app/routes/nova.js` e adicionaremos um método `model` que retornará um objeto com uma propriedade `nome`:

```
//app/routes/nova.js

import Ember from 'ember';

export default Ember.Route.extend({

    model(){
        return Ember.Object.create({nome: null});
    }

});
```

Editaremos também o template da nossa rota `nova`:

```
{{!app/templates/nova.hbs}}
```

```
<label for="nome">Nome</label>
{{input id="nome" value=model.nome}}
```

Então, podemos criar o botão que direcionará para essa nova tela. Ele conterá um atributo `action` e receberá como primeiro parâmetro o nome da ação, nesse caso `nova` :

```
{{!app/templates/lista.hbs}}

<button {{action 'nova'}}>Nova</button>
<br/>

<label for="filtro-nome">Filtrar por nome</label>
{{input id="filtro-nome" value=nome}}
<br/>

<table>
  <thead>
      <tr>
        <th>Nome</th>
      </tr>
  </thead>
  <tbody>
      {{#each model as |pessoa|}}
        <tr>
          <td>
            {{#link-to 'pessoa' pessoa.id}}{{pessoa.nome}}{{/link
-to}}
          </td>
        </tr>
      {{/each}}
  </tbody>
</table>
```

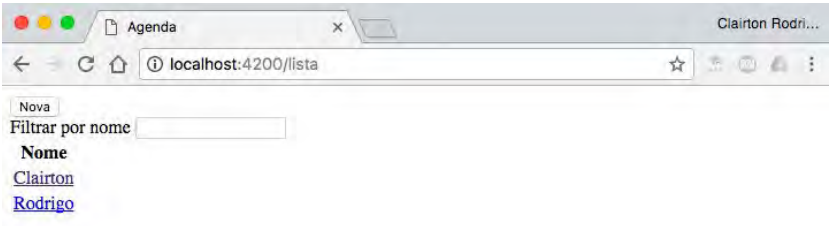

Figura 8.1: Adicionando um botão novo

As ações são funções que devem ficar dentro da propriedade `actions`. Agora precisamos criar essa ação chamada `nova`, dentro da rota `lista`.

```
//app/routes/lista.js

import Ember from 'ember';

export default Ember.Route.extend({
  pessoas: [{id: "1", nome: 'Clairton'}, {id:"2", nome: 'Rodrigo'
}],

  queryParams: {
    nome: {
      refreshModel: true
    }
  },

  model(params){
    return this.get('pessoas').filter((pessoa) => {
      return Ember.isEmpty(params.nome) ||pessoa.nome.indexOf(par
ams.nome) >= 0;
    });
  }

  actions:{

      nova(){
```

```
            this.transitionTo('nova');
        }

    }
});
```

Na ação `nova`, invocamos o método `transitionTo` passando o nome da rota de destino `nova`.

Já temos um botão que direciona para a nova tela, mas precisamos também criar o botão que chamará o código que vai salvar os dados preenchidos dessa nova pessoa. Ele invocará uma ação chamada `salvar`, que criaremos em seguida:

```
{{!app/templates/nova.hbs}}

<label for="nome">Nome</label>
{{input id="nome" value=model.nome}}

<br/>

<button id="salvar" {{action 'salvar' model}}>Salvar</button>
```

Essa nova ação vai ser criada dentro do controlador com o mesmo nome da rota. Como o controlador ainda não existe, vamos criá-lo com o comando:

```
ember generate controller nova
```

Com a ação, ficará assim:

```
//app/controllers/nova.js

import Ember from 'ember';

export default Ember.Controller.extend({
    actions:{
```

```
        salvar(pessoa){
            //veremos como salvar a pessoa mais tarde
            this.transitionToRoute('lista');
        }

    }
});
```

Então, nossa tela para incluir uma nova pessoa ficará assim:

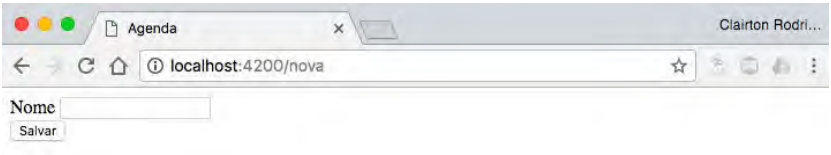

Figura 8.2: Formulário para nova Pessoa

Por padrão, a ação é invocada no evento de `click`, mas é possível alterar esse comportamento utilizando o parâmetro `on`. Para demonstrar isso, vamos criar uma nova ação chamada `salvarAoPressionarEnter`, na qual vamos chamar programaticamente a ação `salvar`.

```
//app/controllers/nova.js

import Ember from 'ember';

export default Ember.Controller.extend({
    actions:{

        salvar(pessoa){
            //veremos como salvar a pessoa mais tarde
            this.transitionToRoute('lista');
        },

        salvarAoPressionarEnter(){
            this.send('salvar', this.get('model'));
```

```
        }

    }
});
```

E agora faremos a invocação da ação que criamos no template:

```
{{!app/templates/nova.hbs}}

<label for="nome">Nome</label>
{{input id="nome" value=model.nome action='salvarAoPressionarEnte
' on='enter'}}

<br/>

<button id="salvar" {{action 'salvar' model}}>Salvar</button>
```

Como podem ver, chamamos a ação `salvar` dentro da ação `salvarAoPressionarEnter`, através do método `send`. Ele espera no primeiro parâmetro o nome da ação que queremos invocar e, em seguida, os parâmetros que essa ação necessita.

Existem muitos outros eventos que podem ser utilizados. Eles podem ser divididos em eventos de teclado, de formulário, de dispositivos sensíveis ao toque, de movimentos do mouse, além dos eventos de arrastar e soltar presentes no HTML5. Veja a seguir uma lista com todas as ações:

- Dispositivos sensíveis ao toque:
    - `touchStart`
    - `touchMove`
    - `touchEnd`
    - `touchCancel`
- Teclado:

E-book gerado especialmente para Matheus Rodrigues F da Silva - matheusuniversal@yahoo.com.br

    - `keyDown`
    - `keyUp`
    - `keyPress`
- Mouse:
    - `mouseDown`
    - `mouseUp`
    - `contextMenu`
    - `click`
    - `doubleClick`
    - `mouseMove`
    - `focusIn`
    - `focusOut`
    - `mouseEnter`
    - `mouseLeave`
- Formulário:
    - `submit`
    - `change`
    - `focusIn`
    - `focusOut`
    - `input`
- Arrastar e soltar do HTML5:
    - `dragStart`
    - `drag`
    - `dragEnter`
    - `dragLeave`
    - `dragOver`
    - `dragEnd`

E-book gerado especialmente para Matheus Rodrigues F da Silva - matheusuniversal@yahoo.com.br

- drop

Eventos podem ser desabilitados ou configurados em nível de aplicação. Por padrão, os eventos do DOM são ouvidos pelo `Ember.EventDispatcher`, e então encaminhados para a `Ember.View`.

```
//app/app.js

var App = Ember.Application.create({
  customEvents: {
    //desabilitando os evento `mouseenter`
    mouseenter: null,
    //adicionando suporte ao evento `paste`
    paste: 'paste'
  }
});
```

Outro comportamento padrão é que o click do mouse é ignorado quando acontece com alguma tecla pressionada. Caso haja necessidade de mudar isso, usamos o parâmetro `allowedKeys`, como no código a seguir:

```
<button {{action "salvar" model  allowedKeys="alt"}}>Salvar</butt
on>
```

Todos os elementos HTML podem receber uma ação, mas alguns não vão mostrar um cursor diferente para o mouse quando ele passar por cima. Para ajustar isso, podemos utilizar o CSS:

```
/*app/styles/app.css*/

[data-ember-action]:not(:disabled) {
  cursor: pointer;
}
```

Até agora, nossa agenda está com os dados fixos. No próximo capítulo, veremos como recuperar as pessoas disponíveis no endereço `http://agenda.clairton.eti.br` .

Capítulo 9

OS MODELOS DA APLICAÇÃO

Analisar e organizar os modelos de negócio sempre é um desafio. Aproximar a modelagem do mundo real em teoria é fácil, mas podemos perder horas e horas toda vez com requisitos não funcionais. Com a ajuda dos modelos do Ember, podemos focar exatamente no que interessa: entregar valor de negócio para nossa aplicação.

Os modelos do Ember, diferente dos controladores e das rotas, estão em uma biblioteca diferente chamada `ember-data`. Esta, por sua vez, já vem instalada por padrão em um projeto criado pelo `ember-cli`.

Em tecnologias back-end, temos várias ferramentas de ORM (*Object Relational Mapping*) que diminuem a impedância com o banco de dados, ou seja, transformam objetos para o modelo relacional e vice-versa. Cada tecnologia tem uma ou mais opções, JPA para o Java, ActiveRecord para o Ruby, NHibernate para .Net, e assim por diante.

Essas bibliotecas de ORM nos ajudam a preocuparmo-nos principalmente com as regras de negócio, focando naquilo que

agrega valor para a aplicação que estávamos desenvolvendo. Podemos dizer que temos mais tempo para focar nos requisitos funcionais das aplicações, enquanto a persistência de dados já tem os moldes prontos.

O `ember-data` tem esse mesmo papel das ferramentas ORM no mundo back-end. Ele cuida da impedância dos dados, transformando objetos em requisições HTTP, e requisições HTTP em objetos. Dessa forma, os dados da aplicação serão salvos e não serão perdidos quando o navegador for fechado.

O `ember-data` possui uma vasta gama de adaptadores dentro do ecossistema desenvolvido pela comunidade, fazendo com que ele se relacione muito bem com vários tipos de back-end. Mais à frente teremos um capítulo abordando especificamente isso. Apesar de fornecer uma base toda sólida, ele exige uma aprendizagem inicial, e é o que faremos aqui.

Podemos definir um modelo como uma classe com atributos que representam algo no mundo real. Por exemplo, em nossa aplicação de agenda que estamos desenvolvendo, ele tem um intuito de substituir uma agenda física, na qual anotamos as pessoas e seus telefones. Ou seja, temos um modelo, a pessoa, que é a representação de uma pessoa que temos na vida real, e esse modelo `Pessoa` tem um atributo que já usamos, o `nome`, e outros que ainda não usamos, o `sobrenome` e também o `nascidoEm`.

Os modelos no Ember ficam localizados em `app/models`, e devem estender da classe `Model` que está no `ember-data`, sendo que o nome do arquivo é o nome do modelo. O modelo `Pessoa` com os atributos `nome`, `sobrenome` e `nascidoEm` pode ser

gerado através do comando:

```
ember generate model pessoa nome:string sobrenome:string nascidoE
m:date
```

```
//app/models/pessoa.js

import DS from 'ember-data';

export default DS.Model.extend({
  nome: DS.attr('string'),
  sobrenome: DS.attr('string'),
  nascidoEm: DS.attr('date')
});
```

No exemplo anterior, criamos vários atributos, entre eles o atributo `nome`, determinando que o tipo dele é `string`, assim como o `sobrenome`, e um atributo `nascidoEm` do tipo `date`. Os atributos podem ser basicamente de quatro tipos. Além do tipo `string` e `date` já vistos, temos os tipos `number` para números inteiros ou não, e o tipo `boolean` para aceitar somente duas respostas, verdadeiro ou falso. Esses tipos estão alinhados com os tipos nativos da linguagem JavaScript.

Os atributos dentro dos modelos são definidos usando o método `DS.attr`, em que o primeiro parâmetro é o tipo de dados que ele comportará. Porém, temos um segundo parâmetro, que é um objeto no formato chave/valor, com algumas possibilidades de customização. Entre essas configurações, temos o `defaultValue`, que define o valor padrão que uma nova instância terá na hora de a criarmos. Por exemplo, quero que o nome da pessoa por padrão seja uma string vazia:

```
//app/models/pessoa.js
```

```
import DS from 'ember-data';

export default DS.Model.extend({
  nome: DS.attr('string', {defaultValue: ''}),
  sobrenome: DS.attr('string'),
  nascidoEm: DS.attr('string')
});
```

Agora podemos terminar a ação de salvar no controlador `nova`. Lá salvaremos os dados que foram preenchidos pelo usuário através do método `save`, e posteriormente redirecionaremos para a página que lista as pessoas, ou seja, para a rota `lista`:

```
//app/controller/nova.js

import Ember from 'ember';

export default Ember.Controller.extend({

    actions:{

        salvar(pessoa){
            let self = this;
            pessoa.save().then(() => {
                self.transitionToRoute('lista');
            });
        }

    }
});
```

Repare que, após a chamada do método `save`, houve a chamada do método `then`. Esta recebe um trecho de código que será executado quando a requisição feita para o back-end retornar com sucesso.

Você pode se perguntar: então a chamada do método `save`

não me retornará a nova pessoa que foi salva? Podemos dizer que não, pois esse método vai retornar uma promessa de salvar os dados da nova pessoa, por isso o uso do método `then`, que é executado quando essa promessa for cumprida com sucesso.

Isso acontece porque o ato de salvar enviará uma requisição AJAX (*Asynchronous JavaScript and XML*), para um determinado endereço em sua rede local ou para a rede mundial de computadores. E quando esse endereço responder que a operação foi concluída com sucesso, o `self.transitionToRoute('lista')` será executado.

Além disso, na rota com o mesmo nome desse controlador, devemos substituir a criação do modelo no método `model`. Dessa vez, invocaremos o método `createRecord` da classe `Store`; esse método recebe como primeiro parâmetro uma string com o nome do modelo, e um segundo no formato JSON com os valores que desejamos popular os atributos. Como ainda não temos nada para preencher, não colocaremos o segundo parâmetro:

```
//app/route/nova.js

import Ember from 'ember';

export default Ember.Route.extend({

    model(){
        return this.get('store').createRecord('pessoa');
    }
});
```

Podemos alterar também a forma de recuperar as pessoas que já foram cadastradas. Isso acontecerá na rota `lista`. Nesse caso, usaremos nosso repositório de registro chamado `store`,

invocando o método `query`:

```
//app/routes/lista.js

import Ember from 'ember';

export default Ember.Route.extend({

  queryParams: {
    nome: {
      refreshModel: true
    }
  },

  model(params){
    let filter = {};
    if(!Ember.isEmpty(params.nome)){
      filter.nome = params.nome;
    }
    return this.get('store').query('pessoa', filter);
  }
});
```

O método `query` espera no primeiro parâmetro o nome do modelo, e no segundo, um objeto no formato JSON. Nesse caso, estou dizendo à `store` para consultar as pessoas cujo atributo `nome` seja igual ao conteúdo da variável `params.nome`. Passamos esse parâmetro somente se ele estiver preenchido.

Também podemos alterar o método `model` da rota `pessoa`, para recuperar no back-end o registro do modelo `Pessoa` pelo `id`:

```
//app/routes/pessoa.js

import Ember from 'ember';

export default Ember.Route.extend({
```

```
    model(params){
        return this.get('store').find('pessoa', params.id);
    }
});
```

Ainda existem bastantes informações sobre os modelos no Ember. Mas por enquanto, o que temos visto é apenas o necessário para conferirmos o papel dos adaptadores no próximo capítulo.

Capítulo 10

CONFIGURE O ADAPTADOR

Já conhecemos os modelos do `ember-data`, e criamos o modelo `Pessoa` contendo os atributos `nome`, `sobrenome` e `nascidoEm`.

```
//app/models/pessoa.js

import DS from 'ember-data';

export default DS.Model.extend({
  nome: DS.attr('string'),
  sobrenome: DS.attr('string'),
  nascidoEm: DS.attr('string')
});
```

Também vimos como criar uma instância desse modelo no método `model` da rota `nova`:

```
//app/routes/nova.js

...

this.get('store').createRecord('pessoa');

...
```

E invocamos o método `save` para persistir os dados na ação

`salvar` da mesma rota.

```
//app/routes/nova.js

...

salvar(pessoa){
  let self = this;
  pessoa.save().then(() => {
      self.transitionTo('lista');
  });
}

...
```

Esse método `save` efetuará uma requisição para um endereço, e é no adaptador que configuraremos esse endereço. Para isso, vamos criar o adaptador padrão da nossa aplicação com o nome `application`.

```
ember generate adapter application
```

E teremos um arquivo semelhante a esse:

```
//app/adapters/application.js

import DS from 'ember-data';

export default DS.JSONAPIAdapter.extend({
});
```

Antes da versão 2 do `ember-data`, o `RESTAdapter` era o adaptador padrão. Já na versão 2, o `JSONAPIAdapter` passou a ser o padrão. A intenção é criar uma especificação para APIs Web. Você pode encontrar mais informações em http://jsonapi.org/, porém ela não funciona em APIs RESTful (*Representational State Transfer*) tradicionais e, se for usada, precisa de muitas alterações

para se integrar.

Utilizaremos o adaptador `RESTAdapter`, pois ele fará as requisições no padrão RESTful onde as operações para manter os dados estão alinhadas aos verbos HTTP. Dessa forma, quando criarmos um novo registro, ele usará o verbo `POST`; para atualizar, o `PUT` ou `PATCH`; para remover, o `DELETE`; e para recuperar, o `GET`.

```
//app/adapters/application.js

import DS from 'ember-data';

export default DS.RESTAdapter.extend({
});
```

Assim sendo, em nosso modelo `pessoa`, teríamos a seguinte relação entre endereço, verbo HTTP, operação e os métodos do `ember-data` que já usamos, levando em consideração o recurso `pessoas`:

Descrição	Verbo HTTP	Método Ember	Endereço
Filtrando as pessoas pelo nome	*GET*	*this.store.query('pessoa', {nome: 'Clairton'})*	*/pessoas?nome=Clairton*
Criando uma nova pessoa	*POST*	*pessoa.save()*	*/pessoas*
Pesquisando a pessoa com *id 123*	*GET*	*this.store.find('pessoa', 123)*	*/pessoas/123*

Além dos métodos do `ember-data` que não utilizamos:

Descrição	Verbo HTTP	Método	URL
Atualizando os dados de			

uma pessoa com o *id 123*	*PUT*	*pessoa.save()*	*/pessoas/123*
Removendo a pessoa com o *id 123*	*DELETE*	*pessoa.destroyRecord()*	*/pessoas/123*

Podemos perceber que o nome do recurso é o nome do modelo no plural, nesse caso `pessoas`. Se houver alguma palavra irregular, ou que a tradução e pluralização do Ember não funcione como esperado, é possível especificar isso através da classe `Inflector` do pacote `ember-inflector`. Geralmente, isso é feito no arquivo `app.js`.

Por exemplo, para especificar a tradução de `aplicacao` para `aplicacoes`, e não `aplicacaos`, faríamos o seguinte:

```
//app/app.js
...
import Inflector from 'ember-inflector';
Inflector.inflector.irregular('aplicacao', 'aplicacoes');
...
```

Para alterar o endereço de determinado modelo, podemos também sobrescrever o método `pathForType` que recebe como parâmetro o tipo em questão. Veja o mesmo exemplo de um modelo chamado `aplicacao`:

```
//app/adapters/application.js
...
export default DS.RESTAdapter.extend({
  ...
  pathForType(type) {
    if(type === 'aplicacao'){
      return 'aplicacoes';
    }
    return type;
  }
  ...
```

```
});
```

A base do endereço com a qual a nossa agenda se conectará está localizada em `http://agenda.clairton.eti.br`. É uma aplicação bem básica escrita em Java e que utiliza o VRaptor, cujo fonte está disponível em `http://github.com/clairton/agenda`.

Precisamos indicar que nossa agenda faça as requisições para esse endereço. Faremos isso através do atributo `host`. Também é necessário setar a propriedade `namespace` para `null`, o que indicará que a API está publicada na raiz:

```
//app/adapters/application.js

import DS from 'ember-data';

export default DS.RESTAdapter.extend({
  host: 'http://agenda.clairton.eti.br',
  namespace: null
});
```

Agora realmente a nossa aplicação escrita com Ember está listando as pessoas. Quando visitamos o navegador em `http://localhost:4200/lista`, conseguimos ver as pessoas cadastradas em nosso back-end. Nessa imagem de tela, deixei a ferramenta de desenvolvedores aberta para ilustrar a requisição efetuada para o back-end:

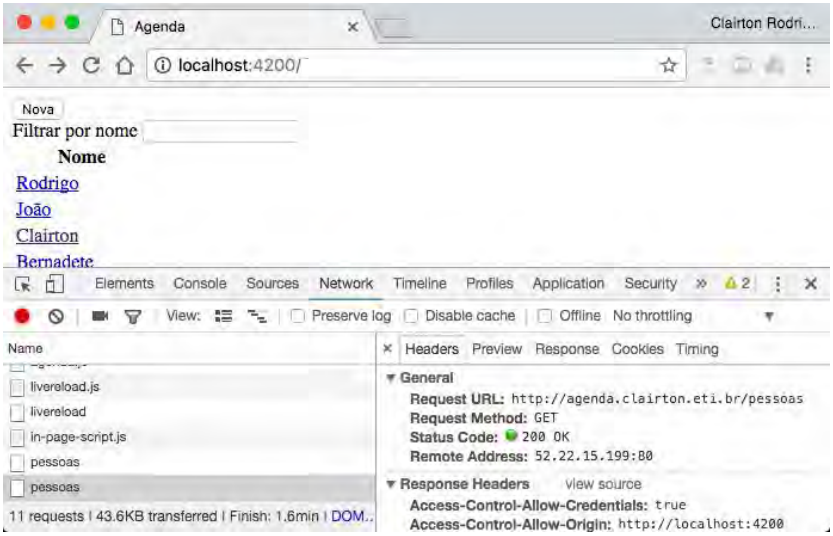

Figura 10.1: Lista de Pessoas vinda do back-end

Do mesmo modo, as outras funcionalidades efetuadas, como a criação e recuperação da pessoa, estarão comunicando com o back-end no endereço `http://agenda.clairton.eti.br` .

Outro adaptador muito conhecido é o *ActiveModelAdapter*. Ele se integra muito bem a aplicações desenvolvidas com o framework Ruby on Rails. Entre suas características, se destaca o fato de transformar o *CamelCase* em *underscore*. Até a versão 2, ele estava presente nativamente, mas agora depende da instalação do add-on `active-model-adapter` .

Existem outros vários adaptadores que são desenvolvidos pela comunidade, mas se ainda assim não encontrar nada que possa ao menos customizar, você pode também desenvolver seu próprio adaptador estendendo de `DS.Adapter` .

Adicionando informações no cabeçalho HTTP

Em alguns momentos, será necessário também passar informações no cabeçalho HTTP. Um exemplo disso seria os dados de autenticação. Podemos fazê-lo da seguinte forma, passando um conjunto de pares chave/valor:

```
//app/adapters/application.js

....
export default DS.RESTAdapter.extend({
  ...
  headers: {
    'CHAVE_API': 'chave secreta',
    'OUTRO_CABECALHO': 'algum outro valor'
  }
  ...
});
```

Os `headers` também podem usar propriedades computadas. Dessa forma, poderá ter valores dinâmicos. Em nosso exemplo, estamos usando o serviço `session` para recuperar as informações.

```
// app/adapters/application.js

export default DS.RESTAdapter.extend({
  ...
  session: Ember.inject.service('session'),

  headers: Ember.computed('session.authToken', function() {
    return {
      'CHAVE_API': this.get('session.authToken')
    };
  })
  ...
});
```

No próximo capítulo, vamos criar um modelo novo para recuperar os telefones.

Capítulo 11

RELACIONANDO PESSOAS COM TELEFONES

Em nossa aplicação de agenda, ainda temos somente o modelo `Pessoa` com o nome. Mas queremos chegar a uma API que mostre os telefones de contato, como temos em `http://agenda.clairton.eti.br/telefones`. Para buscar um telefone lá, basta efetuar uma requisição com o verbo HTTP `GET`, que terá uma resposta semelhante a essa:

```
{
    telefones: [
      {
          id: 1,
          numero: 33333333,
          prefixo: 49
      },{
          id: 2,
          numero: 88888888,
          prefixo: 49
      }
    ]
}
```

Neste capítulo, nosso objetivo é mostrar, além do que já vimos, os respectivos telefones de cada pessoa. Para isso, nosso primeiro passo para buscar esses telefones é criar o modelo com seus atributos:

```
ember generate model telefone numero:number prefixo:number
```

Esse comando gerará um novo arquivo `app/models/telefone.js`, com o conteúdo semelhante a esse:

```
//app/models/telefone.js
import DS from 'ember-data';

export default DS.Model.extend({
  prefixo: DS.attr('number'),
  numero: DS.attr('number'),
});
```

Com o modelo criado, já poderíamos pesquisar os telefones usando a `store`, como por exemplo `this.get('store').query('telefone, {})`. Mas isso não faz muito sentido, pois precisamos relacionar `Pessoa` e `Telefone` para ter uma informação útil.

Então, vamos dizer ao `ember-data` que o modelo `Telefone` pertence a `Pessoa`. Para isso, usaremos o método `DS.belongsTo`, passando como parâmetro o nome do modelo (nesse caso, `pessoa`).

```
//app/models/telefone.js
import DS from 'ember-data';

export default DS.Model.extend({
  prefixo: DS.attr('number'),
  numero: DS.attr('string'),
  pessoa: DS.belongsTo('pessoa')
});
```

Já no caso do modelo `Pessoa`, devemos dizer ao `ember-data` que ele pode possuir vários registros do tipo `Telefone`. Para isso, usaremos o método `DS.hasMany` passando o nome do

modelo `telefone`.

```
//app/models/pessoa.js
import DS from 'ember-data';

export default DS.Model.extend({
  nome: DS.attr('string'),
  sobrenome: DS.attr('string'),
  nascidoEm: DS.attr('string'),
  telefones: DS.hasMany('telefone', {async: true})
});
```

Para mostrar os telefones, vamos editar o componente `mostra-pessoa` e iterar com o helper `each` no atributo `telefones`. Dessa forma, mostraremos todos os telefones:

```
{{!app/components/mostra-pessoa.hbs}}

{{yield}}

Nome: {{pessoa.nome}} - {{pessoa.sobrenome}}
<br/>

Telefones:<br/>
{{#each pessoa.telefones as |telefone|}}
  ({{telefone.prefixo}}) {{telefone.numero}}
  <br/>
{{/each}}
```

Agora visitando o navegador no endereço `http://localhost:4200/lista` e clicando em um dos registros, teremos algo parecido com:

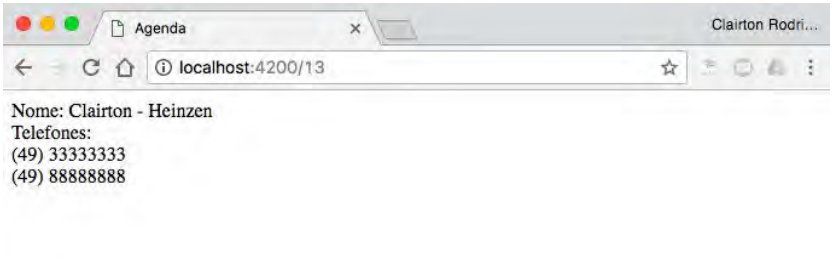

Figura 11.1: Mostrando Pessoa com Telefones

Você pode perceber que passamos um segundo parâmetro `{async: true}` para método `DS.hasMany`. Com isso, estamos dizendo que os `telefones` serão carregados assincronamente. Isso porque quando é feita a requisição ao endereço `http://agenda.clairton.eti.br/pessoas`, dentro da tag `telefones`, há apenas uma coleção com os identificadores dos `telefones` que pertencem à pessoa.

```
{
  pessoas:[
    {
      id: 1,
      nome: 'Clairton',
      sobrenome: 'Heinzen',
      nascidoEm: '1985-04-07',
      telefones: [1, 2]
    }
  ]
}
```

Posteriormente, o `ember-data` se encarregará de realizar as requisições para os endereços `http://agenda.clairton.eti.br/telefones/1` e `http://agenda.clairton.eti.br/telefones/2`, retornando algo como:

```
{
    telefone:{
      id: 1,
      numero: 33333333,
      prefixo: 49
    }
}
```

E para a segunda requisição, para buscar o `telefone`, teremos algo como:

```
{
    telefone: {
      id: 2,
      numero: 88888888,
      prefixo: 49
    }
}
```

Com essas respostas, o `ember-data` vai popular objetos e mostrar na tela.

A quantidade de requisições poderia ser diminuída para 1 se a busca fosse feita com um endereço como esse `http://agenda.clairton.eti.br/telefones?ids[]=1&ids[]=2`. A resposta obtida seria:

```
{
    telefones: [
      {
          id: 1,
          numero: 33333333,
          prefixo: 49
      },{
          id: 2,
          numero: 88888888,
          prefixo: 49
      }
    ]
```

```
}
```

Para isso, precisamos setar a propriedade `coalesceFindRequests` para `true` no adaptador, e o `ember-data` se encarregará do resto:

```
//app/adapters/application.js

import DS from 'ember-data';

export default DS.RESTAdapter.extend({
  host: 'http://agenda.clairton.eti.br',
  namespace: null,
  coalesceFindRequests: true
});
```

A comunicação entre o `ember-data` e o servidor é feita no formato JSON (*JavaScript Object Notation*), e quem determina como isso é processado são os serializadores.

Capítulo 12

PERSONALIZANDO O JSON

Para que a aplicação de agenda que estamos escrevendo consiga se comunicar com a API, que está publicada no endereço `http://agenda.clairton.eti.br` , deve haver um padrão de troca de mensagens. Esse padrão é escrito e lido utilizando o formato JSON.

Quando requisitamos todas as pessoas através da instrução `this.get('store').query('pessoa', {})` , uma requisição é feita para o endereço `http://agenda.clairton.eti.br/pessoas` , como vimos anteriormente. Mas agora, focaremos na resposta que essa requisição vai retornar, que será algo parecido com:

```
{
  pessoas:[
    {
      id: 1,
      nome: 'Clairton',
      sobrenome: 'Heinzen',
      nascidoEm: '1985-04-07'
    }, {
      id: 2,
      nome: 'Rodrigo',
      sobrenome: 'Silva',
```

```
      nascidoEm: '1980-01-31'
    }
  ]
}
```

Alguns detalhes que podemos perceber são: a tag raiz é o nome do modelo que está sendo acessado no plural; e o conteúdo dela é um conjunto de registros, nesse caso, tendo dois objetos, ou seja, duas pessoas.

Para que todos os serializadores de todos os modelos entendam esse formato, criamos o serializador padrão através do comando:

```
ember generate serializer application
```

E especificamos que ele estende da classe `DS.RESTSerializer`:

```
import DS from 'ember-data';

export default DS.RESTSerializer.extend({

});
```

Para retornar apenas os dados da pessoa com um `id` específico, usamos o método da `store`, com algo como `store.findRecord('pessoa', 1)`. Esse método efetuará uma requisição para o endereço `http://agenda.clairton.eti.br/pessoas/1`, que responderá com um JSON similar a esse:

```
{
  pessoa:{
    id: 1,
    nome: 'Clairton',
    sobrenome: 'Heinzen',
```

```
    nascidoEm: '1985-04-07'
  }
}
```

Nesse caso, a tag raiz da resposta é apenas o nome do modelo. Quando houver a criação ou a atualização de um registro de pessoa, o JSON deve seguir o mesmo formato.

É esperado que cada representação contenha uma tag com o identificador do objeto. Por padrão, essa tag se chama `id`, mas isso pode ser alterado caso seja necessário, utilizando o parâmetro de configuração `primaryKey`. Caso o identificador tivesse dentro da tag `id_pessoa`:

```
import Applicaton from '../serializer/application';

export default Applicaton.extend({
    primaryKey: 'id_pessoa'
});
```

Cada atributo da classe `pessoa` será serializado para uma tag que tem como chave, por padrão, o nome do atributo. Porém, isso pode ser alterado usando o método `keyForAttribute`, que recebe como parâmetro o nome do atributo, e retorna o texto que será usado como tag. O exemplo a seguir mostra como o atributo `nascidoEm` seria serializado para tag `nascido_em`:

```
import Applicaton from '../serializer/application';

export default Applicaton.extend({

    keyForAttribute(attribute){
        if(attribute === 'nascidoEm'){
            return 'nascido_em';
        } else {
            return attribute;
        }
```

```
    }

});
```

Como podemos ver, esse método é chamado por todos os atributos e, mediante o valor recebido como parâmetro, podemos determinar o retorno. Algo mais simples poderia ser feito e teria o mesmo resultado, usando a tag `attrs` com uma chave. Esta teria o nome do atributo com o valor da tag pretendida.

```
import Applicaton from '../serializer/application';

export default Applicaton.extend({

    attrs:{
         nascidoEm: 'nascido_em'
    }

});
```

É possível também ignorar um atributo na serialização. Para isso, colocamos o nome do atributo dentro da tag `attrs` , com o conteúdo `serialize: false` :

```
import Applicaton from '../serializer/application';

export default Applicaton.extend({

    attrs:{
         nascidoEm: {
              serialize: false
         }
    }

});
```

Podemos também ignorá-lo na desserialização. Isto é, quando o conteúdo da resposta é lido, isso é feito no parâmetro de

configuração `deserialize: false`.

```
import Applicaton from '../serializer/application';

export default Applicaton.extend({

  attrs:{
    nascidoEm: {
      deserialize: false
    }
  }

});
```

Também é possível fazer desta maneira:

```
//app/serializers/pessoa.js

import DS from 'ember-data';
import Application from '../application';
export default Application.extend(DS.EmbeddedRecordsMixin,{
  attrs: {
    telefones: {
      deserialize: 'ids',
      serialize: 'records'
    }
  }
});
```

É desnecessário enviar o atributo `pessoa` presente em `telefone`, então geramos o serializador para `telefone` e o configuramos para desconsiderar o atributo `pessoa`.

```
//app/serializers/telefone.js
import DS from 'ember-data';
import Application from '../application';
export default Application.extend(DS.EmbeddedRecordsMixin,{
  attrs: {
    pessoa: {
      serialize: false
    }
```

```
  }
});
```

Agora, apesar de termos os telefones listados, eles estão com o número sem formatação, tornando difícil para a percepção humana. Por exemplo, o número de telefone 3333-3333 no código seria exibido como 33333333. Veremos uma maneira de resolver isso no próximo capítulo.

Capítulo 13

CRIANDO UM TIPO DE ATRIBUTOS - TRANSFORMADORES

Já mencionamos que, por padrão, o `ember-data` aceita somente os tipos `string`, `boolean`, `number` e `date`. Sabemos também que esses tipos são passados como parâmetro para o método `DS.attr` nos atributos dos modelos. Além desses tipos padrões, podemos criar nossos próprios tipos de atributos, usando o conceito de transformadores.

Voltando aos telefones, estamos recebendo um número sem estar formatado, como por exemplo `33333333`. Seria mais interessante exibi-lo como `3333 3333`, tornando-o mais legível para o ser humano. É isso que vamos fazer usando um transformador.

O primeiro passo é usar o gerador do `ember-cli`, requisitando a criação de um transformador com o nome de `telefone-numero`:

```
ember generate transform telefone-numero
```

Esse gerador criará um novo arquivo:

```
//app/transforms/telefone-numero.js

import DS from 'ember-data';

expor default DS.Transform.extend({

  serialize(deserialized) {
      return deserialized;
  },

  deserialize(serialized) {
      return serialized;
  }

});
```

Um transformador tem dois métodos. Um será o `serialize` que, no nosso caso, recebe o valor formatado e devolve somente os números. Outro é o método `deserialize`, que recebe um parâmetro somente com os números, e devolve o número de telefone formatado. Faremos isso utilizando expressões regulares:

```
//app/transforms/telefone-numero.js

import DS from 'ember-data';

export default DS.Transform.extend({

  serialize(deserialized) {
      return deserialized.replace(/\D/g, '');
  },

  deserialize(serialized) {
      return serialized.toString().replace(/(\d{4})(\d{4})/, '$1
$2');
  }

});
```

Agora precisamos usá-lo no atributo `numero` do modelo

`telefone`, no lugar do atual `number`, que ficará dessa forma:

```
//app/models/telefone.js

import DS from 'ember-data';

export default DS.Model.extend({
  prefixo: DS.attr('number'),
  numero: DS.attr('telefone-numero'),
  pessoa: DS.belongsTo('pessoa')
});
```

Agora, olhando a tela que exibe as informações da pessoa, teremos:

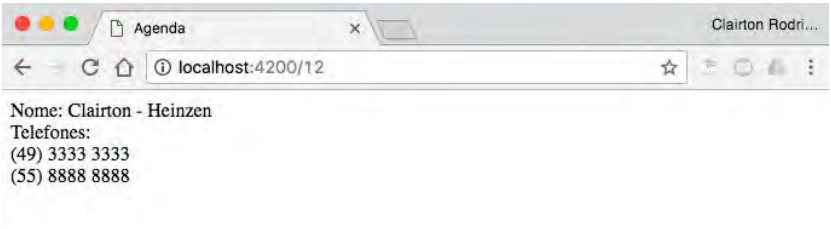

Figura 13.1: Números de telefones formatados

Os transformadores fazem parte da serialização e nos ajudam também a reutilizar código. Por exemplo, o `telefone-numero` usado no atributo `numero` do modelo `telefone` pode ser usado em qualquer outro atributo que se ache necessário:

```
...
  algumAtributo: DS.attr('telefone-numero')
...
```

No próximo capítulo, criaremos um helper, outro recurso interessante do Ember.

Capítulo 14

CRIANDO UM HELPER PARA EXIBIR A DATA POR EXTENSO

Até agora, ainda não usamos o atributo `nascidoEm` do modelo `Pessoa`. Vamos colocá-lo no template do componente `mostra-pessoa`:

```
{{!app/components/mostra-pessoa.hbs}}

{{yield}}

Nome: {{pessoa.nome}} - {{pessoa.sobrenome}}
<br/>

Aniversário: {{pessoa.nascidoEm}}
<br/>

{{#each pessoa.telefones as |telefone|}}
    ({{telefone.prefixo}}) {{telefone.numero}}
    <br/>
{{/each}}
```

O atributo `nascidoEm` é do tipo `date` no padrão ISO, e espera a data no formato `aaaa-mm-dd`. Ele vai imprimir algo como `Dom Apr 07 1985 21:00:00 GMT-0300 (BRT)`, o que é um tanto complicado para entender. O ideal seria mostrar algo

como `07/04/1985`.

Os helpers são soluções ideais para essa situação. Já vimos alguns helpers nativos, como o `if` e o `each`, agora vamos criar o nosso próprio helper para formatar uma data. O `ember-cli` possui um gerador para isso:

```
ember generate helper formata-data
```

Com isso, teremos um arquivo criado em `app/helpers/formata-data.js`:

```
//app/helpers/formata-data.js

import Ember from 'ember';

export function formataData(params/*, hash*/) {
  return params;
}

export default Ember.Helper.helper(formataData);
```

E para usarmos esse helper em nosso template, vamos invocá-lo da seguinte forma:

```
{{formata-data pessoa.nascidoEm}}
```

No componente `mostra-pessoa`, vamos exibir essa informação de data de nascimento somente se ela estiver preenchida. Para isso, usaremos um `if` e, dentro dele, o helper `formata-data`:

```
{{!app/components/mostra-pessoa.hbs}}

{{yield}}
```

```
Nome: {{pessoa.nome}} - {{pessoa.sobrenome}}
<br/>

{{#if pessoa.nascidoEm}}
    Aniversário: {{formata-data pessoa.nascidoEm}}
    <br/>
{{/if}}

{{#each pessoa.telefones as |telefone|}}
    ({{telefone.prefixo}}) {{telefone.numero}}
    <br/>
{{/each}}
```

Veja que passamos como parâmetro a data de nascimento da pessoa. A função JavaScript `formataData` recebe um array chamado `params`, cujo primeiro item é a data. Se por acaso tivéssemos mais parâmetros, eles seriam os próximos itens desse array `params`. Utilizando o objeto do tipo `Date` recebido, vamos retornar o texto da forma que desejamos:

```
//app/helpers/formata-data.js

import Ember from 'ember';

export function formataData(params/*, hash*/) {
  let [nascidoEm] = params;
  let dia = nascidoEm.getDate();
  let mes = nascidoEm.getMonth() +1;
  let ano = nascidoEm.getFullYear();
  return `${dia}/${mes}/${ano}`;
}

export default Ember.Helper.helper(formataData);
```

Note que somamos 1 ao mês, pois ele inicia em 0 - o primeiro índice de um array nas linguagens que se baseiam no `C` -, e termina no 11.

Dessa forma, já teremos uma saída parecida com essa

`07/04/1985` , como podemos ver na figura a seguir:

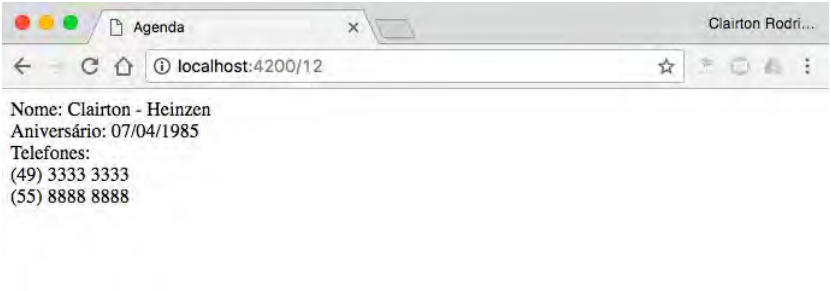

Figura 14.1: Pessoa com aniversário

Ainda mais interessante seria mostrarmos o mês por extenso, em um formato assim `07 de Abril de 1985` . Vamos aproveitar isso para aprender o conceito de serviços e inicializadores nos próximos capítulos.

Capítulo 15

DISPONIBILIZANDO MESES DO ANO POR EXTENSO ATRAVÉS DE UM SERVIÇO

Já utilizamos um serviço do Ember, o objeto `store` , com o qual recuperamos e salvamos dados de uma forma transparente. Mas agora vamos criar nosso próprio serviço, que receberá um valor de 0 a 11, e retornará o mês por extenso, sendo que 0 é Janeiro e 11 é Dezembro.

O funcionamento do Ember internamente baseia-se em um *container*, no qual as dependências são injetadas. No caso dos serviços, a injeção deles é declarativa, ou seja, se preciso de um serviço, preciso declarar essa necessidade. Isso facilita a configuração e a partilha de estado entre os utilizadores (controladores, rotas, componentes e até mesmo outros serviços). Entre os exemplos de uso de serviços, poderíamos destacar o uso de geolocalização e integração com APIs de terceiros.

Para criar nosso serviço, o `ember-cli` possui um gerador, sendo que nosso serviço de tradução para meses por extenso se

suniversal@yah

chamará `meses`. Temos o seguinte comando para executá-lo:

```
ember generate service meses
```

Teremos um arquivo semelhante a esse:

```
//app/services/meses.js

import Ember from 'ember';

export default Ember.Service.extend({
});
```

Como podemos ver, os serviços se localizam na pasta `app/services` e herdam da classe `Ember.service`.

Adicionaremos um método chamado `getPorExtenso`, que recebe como parâmetro o mês em sua representação ordinal de 0 a 11, e devolve o equivalente por extenso ao consultar um array de valores previamente definidos.

```
//app/services/meses.js

import Ember from 'ember';

export default Ember.Service.extend({
  meses: [
    'Janeiro',
    'Fevereiro',
    'Março',
    'Abril',
    'Maio',
    'Junho',
    'Julho',
    'Agosto',
    'Setembro',
    'Outubro',
    'Novembro',
    'Dezembro'
```

```
  ],

  getPorExtenso(ordinal){
    return this.get('meses')[ordinal];
  }
});
```

Para injetar esse serviço em qualquer controlador, rota, componente ou até mesmo outro serviço, declaramos um atributo que receberá o resultado da invocação do método `Ember.inject.service` , que recebe como parâmetro o nome do serviço. Se fôssemos injetar em um controlador:

```
import Ember from 'ember';

export default Ember.Controller.extend({

    meses: Ember.inject.service('meses')

    ...
});
```

Os helpers têm uma particularidade. Para usarmos a injeção de dependência neles, precisaremos estender de `Ember.Helper` e sobrescrever o método `compute` . Então, nosso helper `formata-data` ficará assim:

```
//app/helpers/formata-data.js

import Ember from 'ember';

export default Ember.Helper.extend({

  meses: Ember.inject.service('meses'),

  compute(params/*, hash*/){
    let [nascidoEm] = params;
    let dia = nascidoEm.getDate();
    let mes = this.get('meses').getPorExtenso(nascidoEm.getMonth(
```


suniversal@yah

```
));
    let ano = nascidoEm.getFullYear();
    return `${dia} de ${mes} de ${ano}`;
  }

});
```

Para que seja injetado o serviço no helper, é possível que você tenha de interromper o processo do `ember serve` e iniciá-lo novamente. Para interromper o processo, aperte a tecla `Control` e `c` simultaneamente.

Agora sim, temos nossa data exibida da forma pretendida: `07 de Abril de 1985`. O mês está fixo no idioma português. Se por acaso uma pessoa de outro idioma acessar nossa agenda, seria interessante mostrar essa data no idioma em que o navegador está configurado, não acha?

No próximo capítulo, conheceremos os inicializadores, essenciais para parametrizar aplicações.

Capítulo 16

ADICIONANDO UMA NOVA ENTRADA NA AGENDA

Até o momento, no formulário do template `app/templates/nova.hbs`, temos apenas o atributo `nome` do modelo `Pessoa`, pois foi criado ainda bem no início do livro. Precisamos adicionar os outros atributos `sobrenome` e `nascidoEm`.

Então, vamos editar o template `app/templates/nova.hbs`, colocando os campos para atributos que faltam:

```
{{!app/templates/nova.hbs}}

<label for="nome">Nome</label>
{{input id="nome" value=model.nome action='salvarAoPressionarEnte
' on='enter'}}

<br/>

<label for="sobrenome">Sobrenome</label>
{{input id="sobrenome" value=model.sobrenome}}

<br/>

<label for="nascidoEm">Data Nascimento</label>
```

```
{{input id="nascidoEm" value=model.nascidoEm}}

<br/>

<button id="salvar" {{action 'salvar' model}}>Salvar</button>
```

Além desses atributos, o modelo `Pessoa` também está relacionado ao modelo `Telefone`, através do atributo `telefones`. Isso sinaliza que uma pessoa pode ter nenhum ou vários telefones para contato. Agora, além de listar os telefones, precisamos também dar a possibilidade de remover e adicionar telefones à medida que for necessário.

Vamos começar listando os telefones e dando a possibilidade de removê-lo. Para remover, criaremos uma ação no controlador, presente no arquivo `app/controllers/nova.js`. Ela se chamará `removerTelefone` e receberá uma instância do modelo `Telefone` como parâmetro. A implementação dessa ação fará a invocação do método `destroyRecord` caso o usuário responda positivamente a uma pergunta:

```
//app/controllers/nova.js

import Ember from 'ember';

export default Ember.Controller.extend({

  actions:{

    salvarAoPressionarEnter(){
      this.send('salvar', this.get('model'));
    },

    salvar(pessoa){
      let self = this;
      pessoa.save().then(() => {
        self.transitionToRoute('lista');
      });
```

```
    },

    removerTelefone(telefone){
      if(confirm('Você está removendo um telefone, deseja continu
ar?')){
        telefone.destroyRecord();
      }
    }

  }
});
```

Agora podemos listar os telefones no template, dando a possibilidade de removermos os telefones:

```
{{!app/templates/nova.hbs}}

<label for="nome">Nome</label>
{{input id="nome" value=model.nome action='salvarAoPressionarEnte
' on='enter'}}

<br/>

<label for="sobrenome">Sobrenome</label>
{{input id="sobrenome" value=model.sobrenome}}

<br/>

<label for="nascidoEm">Data Nascimento</label>
{{input id="nascidoEm" value=model.nascidoEm type='date'}}

<br/>

{{#each model.telefones as |telefone|}}
    ({{telefone.prefixo}}) {{telefone.numero}}
    <button {{action 'removerTelefone' telefone}}>Remover</button>

    <br/>
{{/each}}

<br/>

<button id="salvar" {{action 'salvar' model}}>Salvar</button>
```

Quando o método `destroyRecord` for executado para um telefone com o `id 1`, por exemplo, uma requisição com o verbo HTTP `DELETE` será direcionada ao endereço `http://agenda.clairton.eti.br/telefones/1`. Caso a resposta seja sucesso, esse registro também será eliminado do serviço `store` e da instância do model `Pessoa`, sendo que não aparecerá mais na tela para o usuário.

Agora daremos a opção de adicionar um novo telefone, necessitando de uma instância de `store`, e pediremos que o Ember a injete para nós.

```
store: Ember.inject.service()
```

Também adicionaremos uma ação ao controlador que criará uma nova instância do modelo `Telefone` em um atributo do controlador chamado `telefone`. Essa ação se chamará `criarTelefone`.

```
criarTelefone(){
    let telefone = this.get('store').createRecord('telefone', {})
;
    this.set('telefone', telefone);
}
```

Ela deve ser invocada quando o controlador for instanciado, por isso observaremos o evento `init` do controlador.

```
criarTelefoneAoIniciar: Ember.on('init', function(){
  this.send('criarTelefone');
})
```

Daremos a possibilidade de preencher os atributos `prefixo` e `numero`, e também um botão que invoca a ação

`adicionarTelefone` (que criaremos posteriormente), passando o atributo `telefone` do controlador como parâmetro:

```
{{!app/templates/nova.hbs}}

<label for="nome">Nome</label>
{{input id="nome" value=model.nome action='salvarAoPressionarEnte
' on='enter'}}

<br/>

<label for="sobrenome">Sobrenome</label>
{{input id="sobrenome" value=model.sobrenome}}

<br/>

<label for="nascidoEm">Data Nascimento</label>
{{input id="nascidoEm" value=model.nascidoEm type='date'}}

<br/>

<label for="prefixo">Prefixo</label>
{{input id="prefixo" value=telefone.prefixo type='number'}}
<label for="numero">número</label>
{{input id="numero" value=telefone.numero type='number'}}
<button {{action 'adicionarTelefone' telefone}}>Adicionar</button


<br/>

{{#each model.telefones as |telefone|}}
    ({{telefone.prefixo}}) {{telefone.numero}}
    <button {{action 'removerTelefone' telefone}}>Remover</button

    <br/>
{{/each}}


<button id="salvar" {{action 'salvar' model}}>Salvar</button>
```

Nossa tela ficará parecida com:

Figura 16.1: Formulário de Pessoa com Telefone

Na implementação da ação `adicionarTelefone`, adicionaremos o telefone à lista de telefones da pessoa. Também vamos popular o atributo `telefone` do controlador novamente com uma nova instância do modelo `Telefone`, invocando a ação `criarTelefone`:

```
adicionarTelefone(telefone){
  this.get('model.telefones').pushObject(telefone);
  this.send('criarTelefone');
}
```

Nosso controlador do arquivo `app/controller/nova.js` ficará assim:

```
//app/controllers/nova.js

import Ember from 'ember';

export default Ember.Controller.extend({

  store: Ember.inject.service(),

  criarTelefoneAoIniciar: Ember.on('init', function(){
```

```
    this.send('criarTelefone');
  }),

  actions:{

    salvarAoPressionarEnter(){
      this.send('salvar', this.get('model'));
    },

    salvar(pessoa){
      let self = this;
      pessoa.save().then(() => {
        self.transitionToRoute('lista');
      });
    },

    removerTelefone(telefone){
      if(confirm('Você está removendo um telefone, deseja continu
ar?')){
        telefone.destroyRecord();
      }
    },

    criarTelefone(){
        let telefone = this.get('store').createRecord('telefone',
 {});
        this.set('telefone', telefone);
    },

    adicionarTelefone(telefone){
      this.get('model.telefones').pushObject(telefone);
      this.send('criarTelefone');
    }

  }

});
```

Ainda há uma mudança a ser feita quanto à serialização do modelo `Pessoa` . Isso porque, na criação e atualização de dados, nosso back-end espera o JSON dos telefones embutidos dentro do JSON da pessoa, como no exemplo seguinte:

```
pessoa: {
  id: 1,
  nome: 'Clairton',
  sobrenome: 'Heinzen',
  nascidoEm: '1985-04-07',
  telefones:[
    {
      prefixo: 49,
      numero: 88888888
    }
  ]
}
```

Como ainda não criamos nenhum serializador com o mesmo nome do modelo (nesse caso `Pessoa`), o `ember-data` está usando o serializador principal que está no arquivo `app/serializers/application.js`. Para personalizar o JSON do model `Pessoa`, primeiramente devemos criá-lo:

```
ember generate serializer pessoa
```

Por padrão, ele herdará o comportamento do serializador principal e vamos deixá-lo assim:

```
//app/serializers/telefones.js

import DS from 'ember-data';
import ApplicationSerializer from './application';

export default ApplicationSerializer.extend(DS.EmbeddedRecordsMix
in,{

  attrs: {
    telefones: {
      serialize: 'records',
      deserialize: 'ids'
    }
  }
```

```
});
```

Para serializar os telefones de forma embutida em `pessoa`, é necessário que o seu serializador estenda do mixin `DS.EmbeddedRecordsMixin` e que também use o `serialize: 'records'`. Com isso, ao enviar os dados para o back-end, os telefones serão serializados e embutidos dentro do JSON da pessoa. Assim, ao ler do back-end, continuará da forma que usamos até o momento, somente com os identificadores, e posteriormente efetuando a busca pelos telefones com esse identificador.

Já temos a opção de criar um novo registro para a nossa agenda, porém ter a possibilidade da alteração dos dados de um registro já criado também é importante. Esse é o tema do próximo capítulo, no qual basicamente a edição usará as mesmas lógicas da criação. Por isso, tiraremos as implementações do controlador `nova` e isolaremos em um componente, assim sendo possível sua reutilização para edição.

Capítulo 17

DEFININDO UM IDIOMA NA INICIALIZAÇÃO DA AGENDA

Os navegadores de internet possuem uma propriedade no objeto global `navigator` chamada `language`, que, no caso de português do Brasil, retorna `pt-BR` ou, no caso do inglês dos Estados Unidos, retornará `en-US`.

Vamos criar essas duas opções, a princípio, dentro do atributo `meses` no serviço `meses`, dessa forma:

```
meses: {
  'pt-BR': [
    'Janeiro',
    'Fevereiro',
    'Março',
    'Abril',
    'Maio',
    'Junho',
    'Julho',
    'Agosto',
    'Setembro',
    'Outubro',
    'Novembro',
    'Dezembro'
  ],
  'en-US': [
```

```
    'January',
    'February',
    'March',
    'April',
    'May',
    'June',
    'July',
    'August',
    'September',
    'October',
    'November',
    'December'
  ]
},
```

E no método `getPorExtenso`, recuperaremos o mês baseado em um novo atributo que chamaremos de `idioma`:

```
getPorExtenso(ordinal){
  let idioma = this.get('idioma');
  return this.get('meses').get(idioma)[ordinal];
}
```

Também declararemos o atributo `idioma` com o valor padrão de `pt-BR`:

```
idioma: 'pt-BR'
```

Então, nosso serviço deve ficar assim:

```
//app/services/meses.js

import Ember from 'ember';

export default Ember.Service.extend({
  idioma: 'pt-BR',
  meses: {
    'pt-BR': [
      'Janeiro',
      'Fevereiro',
```

```
      'Março',
      'Abril',
      'Maio',
      'Junho',
      'Julho',
      'Agosto',
      'Setembro',
      'Outubro',
      'Novembro',
      'Dezembro'
    ],
    'en-US': [
      'January',
      'February',
      'March',
      'April',
      'May',
      'June',
      'July',
      'August',
      'September',
      'October',
      'November',
      'December'
    ]
  },

  getPorExtenso(ordinal){
    let idioma = this.get('idioma');
    return this.get('meses')[idioma][ordinal];
  }
});
```

Agora precisamos injetar o valor que temos no navegador no atributo `idioma`. Isso deve ser feito quando a aplicação `agenda` que estamos desenvolvendo com o Ember terminar de carregar, ou seja, quando ela iniciar. O Ember mais uma vez disponibiliza uma forma de resolvermos isso: os inicializadores.

Existem dois tipos de inicializadores: o de aplicação e o de instância. No primeiro, temos acesso às configurações com as

quais interagimos antes de a instância ser criada, enquanto o inicializador de instância é executado em um momento posterior. O que usaremos é o de aplicação, pois estaremos criando pontos de injeção entre objetos.

A Injeção de Dependência (ou DI, uma abreviação do termo em *Dependency Injection*) é um Design Pattern usado para instanciar classes de objetos e dependências entre eles. No Ember, ela é baseada em fábricas, que são textos que identificam cada rota, controlador, componente etc. Por exemplo, na nossa rota `lista`, sua fábrica é o texto `route:lista`, enquanto o componente `mostra-pessoa` tem a fábrica `component:mostra-pessoa`.

Para injetar o idioma no serviço `meses`, teremos de criar uma fábrica, para depois poder usar. Para isso, com a ajuda do gerador `initializer` do `ember-cli`, criamos o inicializador `idioma`:

```
ember generate initializer idioma
```

O arquivo gerado terá uma função JavaScript que recebe como parâmetro um objeto do tipo `Ember.Application`:

```
//app/initializers/idioma.js

export function initialize(/*application*/) {
}


export default {
  name: 'idioma',
  initialize: initialize
};
```

Será necessário descomentar o parâmetro `application`, pois invocaremos o método `register` dele, dizendo que necessitamos

de uma nova fábrica chamada `config:idioma`. Esta conterá o valor recuperado de `navigator.language`.

```
//app/initializers/idioma.js

export function initialize(application) {
  let idioma = navigator.language;
  application.register('config:idioma', idioma, { instantiate: fa
lse });
}


export default {
  name: 'idioma',
  initialize: initialize
};
```

No terceiro parâmetro, passamos `{ instantiate : false}`, que significa que o objeto da fábrica `config:idioma` não será instanciado usando o `new` quando for injetado. Ele seguirá o padrão de projetos Singleton, em que o mesmo objeto é compartilhado entre toda a aplicação.

Ainda falta injetar esse nosso objeto no serviço `meses`, que tem a fábrica `service:meses`. Para fazer isso, usaremos o método `inject`, passando a fábrica na qual desejamos que ocorra a injeção de dependência; em segundo, o nome do atributo em que deve ser injetado o valor; e por fim, a fábrica da configuração de idioma `config:idioma`. Em outras palavras, queremos que o serviço de meses tenha a configuração de idioma injetado no atributo `idioma`:

```
//app/initializers/idioma.js

export function initialize(application) {
  let idioma = navigator.language;
```

```
  application.register('config:idioma', idioma, { instantiate: fa
lse });
  application.inject('service:meses', 'idioma', 'config:idioma');
}


export default {
  name: 'idioma',
  initialize: initialize
};
```

Dessa forma, podemos ter a variação dos meses do ano em dois idiomas, de uma maneira bem produtiva e organizada.

Mas e se um usuário fosse tentar acessar com o idioma espanhol, por exemplo, teríamos de definir um idioma padrão entre aqueles suportados. No caso, vamos usar `pt-BR`:

```
//app/services/meses.js

import Ember from 'ember';

export default Ember.Service.extend({
  idioma: 'pt-BR',
  meses: {
    'pt-BR': [
      'Janeiro',
      'Fevereiro',
      'Março',
      'Abril',
      'Maio',
      'Junho',
      'Julho',
      'Agosto',
      'Setembro',
      'Outubro',
      'Novembro',
      'Dezembro'
    ],
    'en-US': [
      'January',
      'February',
```

```
      'March',
      'April',
      'May',
      'June',
      'July',
      'August',
      'September',
      'October',
      'November',
      'December'
    ]
  },

  getPorExtenso(ordinal){
    let idioma = this.get('idioma');
    let meses = this.get('meses')[idioma];
    if(!meses){
        meses = this.get('meses')['pt-BR'];
    }
    return meses[ordinal];
  }
});
```

Já configuramos a visualização dos dados e, nos próximos capítulos, aprenderemos a criar e atualizar registros.

CAPÍTULO 18

REUTILIZANDO A LÓGICA DE CRIAR PARA EDITAR UMA PESSOA

Para reutilizar a lógica de criação de uma pessoa na edição, criaremos um componente chamado `formulario-pessoa` :

```
ember generate component formulario-pessoa
```

Vamos esperar que esse componente receba como parâmetro a instância do modelo `Pessoa` , chamado `pessoa` . Copiaremos o conteúdo do template do arquivo `app/templates/pessoas/nova.hbs` para `app/templates/components/formulario-pessoa.hbs` , substituindo a referência de `model` para `pessoa` :

```
{{!app/templates/componets/formulario-pessoa.hbs}}

<label for="nome">Nome</label>
{{input id="nome" value=pessoa.nome action='salvarAoPressionarEnt
er' on='enter'}}

<br/>

<label for="sobrenome">Sobrenome</label>
{{input id="sobrenome" value=pessoa.sobrenome}}
```

```
<br/>

<label for="nascidoEm">Data Nascimento</label>
{{input id="nascidoEm" value=pessoa.nascidoEm type='date'}}

<br/>

<label for="prefixo">Prefixo</label>
{{input id="prefixo" value=telefone.prefixo type='number'}}
<label for="numero">número</label>
{{input id="numero" value=telefone.numero type='number'}}
<button {{action 'adicionarTelefone' telefone}}>Adicionar</button>


<br/>

{{#each pessoa.telefones as |telefone|}}
  ({{telefone.prefixo}}) {{telefone.numero}}
  <button {{action 'removerTelefone' telefone}}>Remover</button>
  <br/>
{{/each}}


<button id="salvar" {{action 'salvar' pessoa}}>Salvar</button>
```

E o conteúdo do template `app/templates/nova.hbs` chamará o componente `formulario-pessoa`, passando o seu atributo `model` para o parâmetro `pessoa` do componente.

```
{{!app/templates/pessoas/nova.hbs}}

{{formulario-pessoa pessoa=model}}
```

Se você clicar agora nos botões, receberá um erro de ações não encontradas. Isso acontece porque necessitamos mover todas as nossas implementações do controlador `nova`, que estão no arquivo `app/controllers/nova.js`, para o arquivo do componente que está em `app/components/formulario-pessoa.js`:

```
//app/components/formulario-pessoa.js

import Ember from 'ember';

export default Ember.Component.extend({

  store: Ember.inject.service(),

  criarTelefoneAoIniciar: Ember.on('init', function(){
    this.send('criarTelefone');
  }),

  actions:{

    salvarAoPressionarEnter(){
      this.send('salvar', this.get('pessoa'));
    },

    salvar(pessoa){
      let self = this;
      pessoa.save().then(() => {
        self.transitionToRoute('lista');
      });
    },

    removerTelefone(telefone){
      if(confirm('Você está removendo um telefone, deseja continu
ar?')){
        telefone.destroyRecord();
      }
    },

    criarTelefone(){
        let telefone = this.get('store').createRecord('telefone',
 {});
        this.set('telefone', telefone);
    },

    adicionarTelefone(telefone){
      this.get('pessoa.telefones').pushObject(telefone);
      this.send('criarTelefone');
    }

  }
```

```
});
```

E o controlador fica apenas com a declaração:

```
//app/controllers/nova.js

import Ember from 'ember';

export default Ember.Controller.extend({

});
```

Porém, ao salvarmos a pessoa, um erro ocorrerá, dizendo que o método `transitionToRoute` não existe. Isso acontece pelo motivo de os componentes não terem acesso ao navegador entre as rotas. Para resolver esse problema, criamos uma ação no controlador que direciona para a rota `lista`:

```
//app/controllers/nova.js

import Ember from 'ember';

export default Ember.Controller.extend({

  actions:{
      voltaParaLista(){
        this.transitionToRoute('lista');
      }
  }
});
```

Agora a passamos como parâmetro no momento que chamamos o componente, nomeando o parâmetro como `aoSalvarComSucesso`:

```
{{!app/templates/pessoas/nova.hbs}}
```

```
{{formulario-pessoa pessoa=model aoSalvarComSucesso=(action 'volt
aParaLista')}}
```

Vamos substituir a chamada de `transitionToRoute` pela chamada desse parâmetro recebido. Normalmente, para chamar uma ação que se encontra dentro do mesmo controlador, componente ou rota, usamos o `send`, mas como a recebemos como parâmetro, precisamos usar o `sendAction`:

```
//app/components/formulario-pessoa.js
...
actions: {
...
    salvar(pessoa){
      let self = this;
      pessoa.save().then(() => {
        self.sendAction('aoSalvarComSucesso');
      });
    }
...
}
...
```

Feito isso, é possível reutilizarmos o componente `formulario-pessoa` na edição. Começamos criando uma rota para a edição com o comando:

```
ember generate route editar
```

No arquivo `app/router.js`, vamos adicionar um parâmetro com o `id` da pessoa que se deseja editar:

```
//app/router.js

import Ember from 'ember';

export default Ember.Router.map(function() {
```

```
    this.route('lista');
    this.route('nova');
    this.route('pessoa', {path: ':id'});
    this.route('editar', {path: 'editar/:id'});

});
```

E ainda implementaremos o método `model` que vai recuperar a pessoa através do `id` usando o serviço `store`:

```
//app/routes/editar.js

import Ember from 'ember';

export default Ember.Route.extend({

  model(params){

    return this.get('store').findRecord('pessoa', params.id);

  }

});
```

Também vamos implementar o método que direciona para a lista de pessoas no controlador que geramos com o comando:

```
ember generate controller editar
```

```
//app/controllers/editar.js

import Ember from 'ember';

export default Ember.Controller.extend({

  actions:{
      voltaParaLista(){
        this.transitionToRoute('lista');
      }
  }
```

```
});
```

O template localizado em `app/templates/editar.hbs` ficará da seguinte forma:

```
{{!app/templates/editar.hbs}}

{{formulario pessoa=model aoSalvarComSucesso=(action 'voltaParaLi
sta')}}
```

Também daremos a opção de editar a pessoa quando vemos os dados dela, por isso vamos pôr um link em `app/templates/pessoa.hbs` .

```
{{!app/templates/pessoa.hbs}}

{{mostra-pessoa pessoa=model}}

<br/>

{{#link-to 'editar' model.id}}Editar{{/link-to}}
```

Ficando assim:

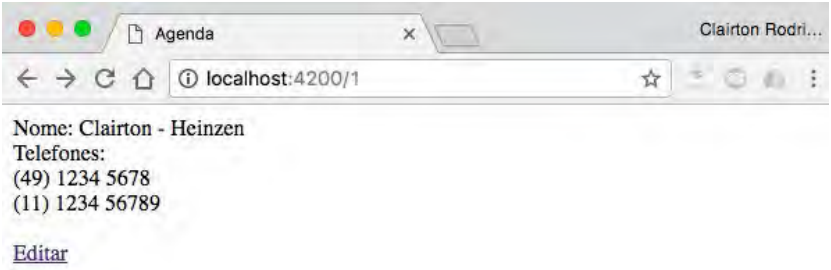

Figura 18.1: Opção de edição

Ao clicar em editar, você poderá perceber, como na imagem a seguir, que a tela de edição está muito semelhante à tela de criação de um novo registro, com exceção do endereço que terá. Nesse caso, o final `/editar/1` .

Figura 18.2: Formulário de edição

Veja que apenas foram usados os geradores de código repetitivo do `ember-cli` , e depois definimos uma ação e usamos o componente já criado. Temos a edição de uma pessoa reutilizando o mesmo formulário da criação, ou seja, componentizamos uma parte de código, para então reusar. Mas ainda há um princípio que não aprendemos no uso do Ember, a boa prática dos testes escritos, que vamos abordar no próximo capítulo.

Capítulo 19

TESTANDO A CRIAÇÃO DE UMA NOVA PESSOA

Os testes têm sido um ponto de grande atenção da comunidade de desenvolvedores em geral, e isso não apenas no Ember. Os testes escritos nos dão segurança para evoluir o software, pois se alterarmos algum comportamento já assegurado por um teste, teremos o feedback de que algo está errado.

Existem vários tipos de testes. O primeiro que vamos escrever simula a interação que um usuário teria com a aplicação, clicando em botões e preenchendo dados. Esse tipo de teste é chamado de teste de aceitação.

Vamos usar o gerador `acceptance-test` para criar um teste simulando a inclusão de uma nova pessoa:

```
ember generate acceptance-test nova
```

Esse comando criará o arquivo `tests/acceptance/nova-test.js` que deve ser semelhante a isso:

```
//tests/acceptance/nova-test.js

import { test } from 'qunit';
import moduleForAcceptance from 'agenda/tests/helpers/module-for-
```

```
acceptance';

moduleForAcceptance('Acceptance | nova');

test('visiting /nova', function(assert) {
  visit('/nova');

  andThen(function() {
    assert.equal(currentURL(), '/nova');
  });
});
```

Visualizamos nesse teste o helper chamado `moduleForAcceptance`, que carrega todo o código que escrevemos para poder executar os testes. O parâmetro passado para ele é o nome do grupo de testes. Depois temos o helper `test`, para o qual passamos o nome do teste, e sua implementação, que alteraremos para `Criar Pessoa`:

```
//tests/acceptance/nova-test.js

import { test } from 'qunit';
import moduleForAcceptance from 'agenda/tests/helpers/module-for-
acceptance';

moduleForAcceptance('Acceptance | pessoas/nova');

test('Criar Pessoa', function(assert) {
  visit('/nova');

  andThen(function() {
    assert.equal(currentURL(), 'nova');
  });

});
```

Por padrão, o Ember utiliza a biblioteca QUnit, já conhecida e utilizada amplamente para fazer testes em aplicações web. No QUnit, a assinatura básica é composta pela invocação de `test`,

passando um nome e um callback com as instruções do teste:

```
test('Nome do Teste', function(assert) {/*implementação do teste*
});
```

Já temos esse mesmo esqueleto do QUnit em nosso teste `"Criar Pessoa"`. Dentro do callback, podemos ver o uso do helper `visit`, usado em todo início de teste de aceitação. Ele recebe um parâmetro que é o endereço para o qual ele deve navegar, nesse caso `/nova`.

```
visit('/nova');
```

Na próxima instrução, você encontrará o helper `andThen`. Este, por sua vez, também recebe um callback, que verifica se o endereço visitado é o esperado, através do helper `currentURL`.

Para rodar os testes, podemos visitar o endereço `http://localhost:4200/tests` em nosso navegador. Para isso, devemos estar com o comando executando `ember server`. Para focar no teste atual que estamos fazendo, vamos filtrar por `"Criar Pessoa"`, e o endereço ficará `http://localhost:4200/tests?filter=Criar%20Pessoa`.

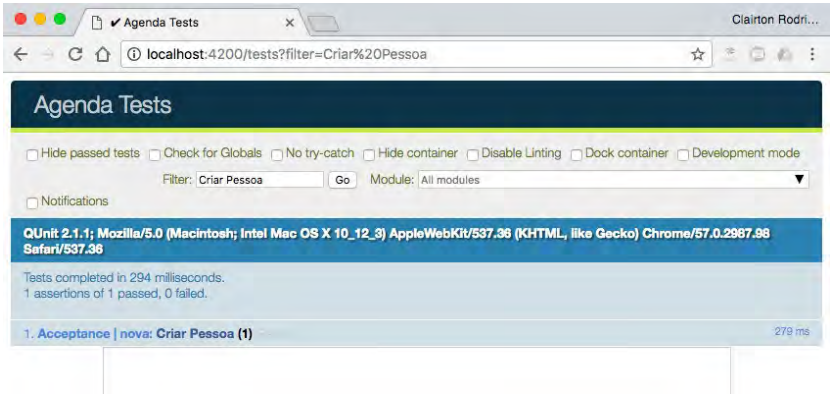

Figura 19.1: Rodando um teste

A instrução `assert.equal` vai comparar o primeiro com o segundo parâmetro recebido. Caso não sejam iguais, uma exceção será lançada, fazendo com que o teste falhe e fique de outra cor:

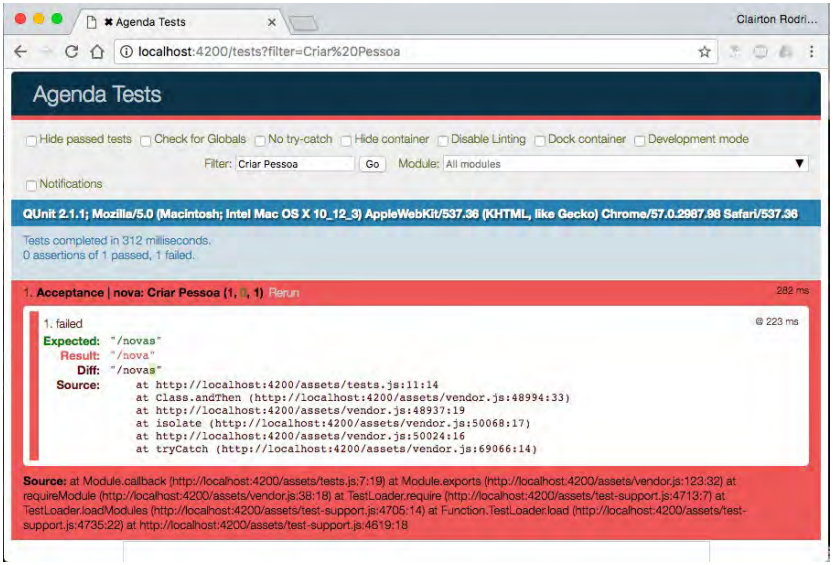

Figura 19.2: Rodando um teste com falha

Mas na maioria das vezes, os testes serão rodados em servidores de integração contínua. Nesse caso, eles podem ser rodados com o comando `ember test`. Todos os testes serão executados. No caso, para rodar o nosso teste que estamos escrevendo, podemos usar o parâmetro `--filter` dessa forma:

```
ember test --filter="Criar Pessoa"
```

Teremos um resultado como:

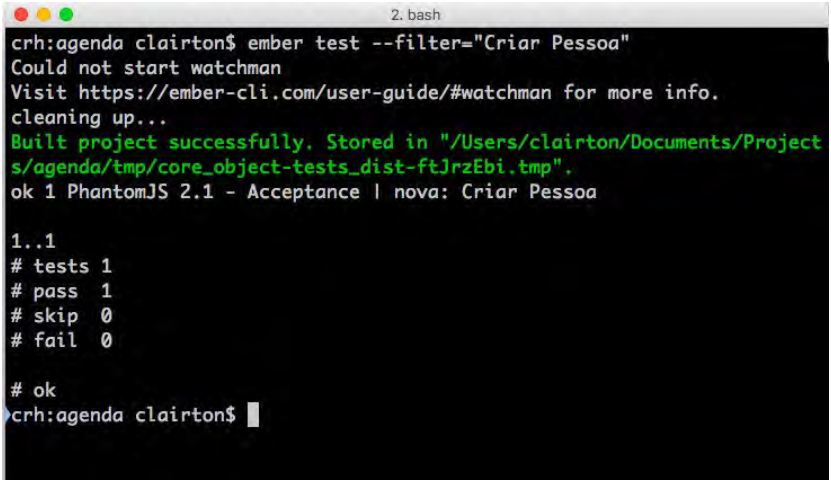

Figura 19.3: Rodando testes no terminal

Na verdade, nosso teste de aceitação ainda não está testando nada de útil, pois ele deveria simular os passos que alguém faria para usar a agenda. Então, vamos agora preencher os campos que esperam o nome, sobrenome e data de nascimento.

Para isso, usaremos o helper `fillIn`. Por exemplo, para preencher o campo `nome` com `Clairton`, ficaria assim:

```
fillIn('#nome', 'Clairton');
```

O helper `fillIn` espera no primeiro parâmetro um seletor que represente o elemento que desejamos preencher e, por segundo, o valor que será colocado. Agora, preencheremos também o sobrenome e a data de nascimento:

```
fillIn('#sobrenome', 'Heinzen');
fillIn('#nascidoEm', '07/04/1985');
```

Após preencher os dados, precisamos clicar no botão com o id `salvar`. Para isso, usaremos o helper `click` passando também um seletor.

```
click('#salvar');
```

Nosso teste ficará no arquivo `tests/acceptance/nova-test.js` dessa forma:

```
//tests/acceptance/nova-test.js

import { test } from 'qunit';
import moduleForAcceptance from 'agenda/tests/helpers/module-for-
acceptance';

moduleForAcceptance('Acceptance | nova');

test('Criar Pessoa', function(assert) {
  visit('/nova');

  andThen(function() {
    assert.equal(currentURL(), '/nova');

    fillIn('#nome', 'Clairton');
    fillIn('#sobrenome', 'Heinzen');
    fillIn('#nascidoEm', '07/04/1985');

    click('#salvar');
```

```
  });
});
```

Ao salvar com sucesso, nossa ação direciona para a lista de pessoas. Essa seria nossa última asserção para esse teste:

```
//tests/acceptance/nova-test.js

import { test } from 'qunit';
import moduleForAcceptance from 'agenda/tests/helpers/module-for-
acceptance';

moduleForAcceptance('Acceptance | nova');

test('Criar Pessoa', function(assert) {
  visit('/nova');

  andThen(function() {
    assert.equal(currentURL(), '/nova');

    fillIn('#nome', 'Clairton');
    fillIn('#sobrenome', 'Heinzen');
    fillIn('#nascidoEm', '07/04/1985');

    click('#salvar');

    assert.equal(currentURL(), '/lista');

  });
});
```

Porém, se executarmos os testes agora no navegador, visitando o endereço `http://localhost:4200/tests?filter=Criar%20Pessoa` , haverá uma falha:

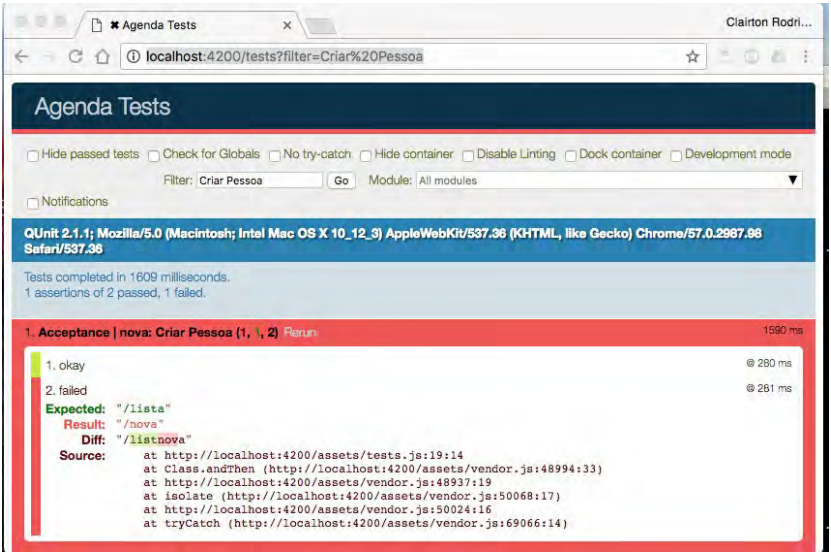

Figura 19.4: Teste mostrando o motivo da falha

O teste está falhando porque espera que o endereço seja `/lista`, mas está sendo `/nova`. Isso acontece porque o processo que inicia ao clicarmos em salvar é assíncrono, e a última asserção não está esperando esse processo de salvar terminar para verificar o endereço. Para executarmos um trecho de código depois do processo terminar, usamos o helper `andThen`. Já o usamos em nosso teste logo após o `visit`:

```
//tests/acceptance/nova-test.js

...

  visit('/nova');

  andThen(function() {

...
```

E agora usaremos o `andThen` na última asserção:

```
//tests/acceptance/nova-test.js

import { test } from 'qunit';
import moduleForAcceptance from 'agenda/tests/helpers/module-for-
acceptance';

moduleForAcceptance('Acceptance | nova');

test('Criar Pessoa', function(assert) {
  visit('/nova');

  andThen(function() {
    assert.equal(currentURL(), '/nova');

    fillIn('#nome', 'Clairton');
    fillIn('#sobrenome', 'Heinzen');
    fillIn('#nascidoEm', '07/04/1985');

    click('#salvar');

    andThen(() => {
        assert.equal(currentURL(), '/lista');
    });

  });
});
```

Ao rodar os testes novamente, voltaremos a ter sucesso como no início.

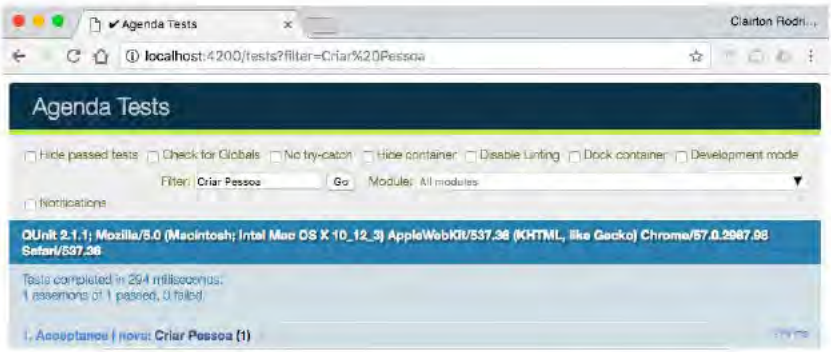

Figura 19.5: Teste passando

A biblioteca usada por padrão para fazer os testes no Ember é o QUnit. A maioria dos helpers que mencionamos vem dela, principalmente as asserções, que comparam os valores obtidos e esperados. O que utilizamos até agora foi o `equal`, que espera que os valores sejam iguais. Porém, existem muitos outros:

- `async`: instrui ao QUnit para esperar por uma operação assíncrona.
- `deepEqual`: tem uma asserção mais profunda e funciona para objetos primitivos, arrays, expressões regulares, datas e funções.
- `expect`: especifica quantas asserções são esperadas no teste.
- `notDeepEqual`: negação do `deepEquals`.
- `notEqual`: negação do `equal`.
- `ok`: espera um resultado positivo.
- `notOk`: negação do `ok`.
- `throws`: testa se determinada exceção foi lançada.

E-book gerado especialmente para Matheus Rodrigues F da Silva - matheusuniversal@yahoo.com.br

Agora, uma pergunta: se não preenchêssemos o campo `nome`, a pessoa teria os dados salvos? A resposta é não. Em nosso back-end, o qual roda no endereço `http://agenda.clairton.eti.br`, existe uma validação, e o teste falharia caso o campo não fosse preenchido:

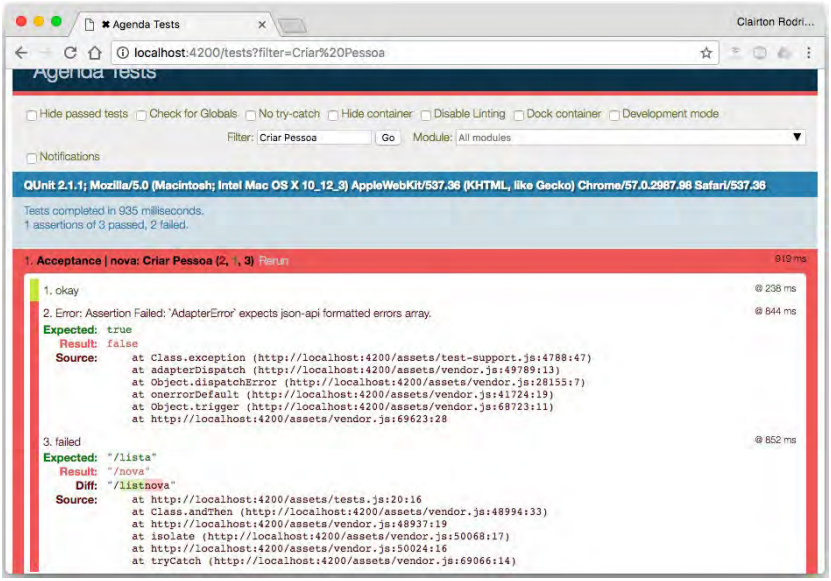

Figura 19.6: Falha na criação

Esse é o tema do próximo capítulo.

Capítulo 20

TRATANDO OS ERROS DE VALIDAÇÃO DEFINIDOS NO SERVIDOR

Quando submetemos os dados da pessoa para salvar, sem enviar o atributo `nome`, a requisição responderá com o código HTTP 422. Isso significa que os dados submetidos estão inválidos, e a resposta em JSON será com o formato parecido com:

```
{
    "errors":{
        "nome":["não pode ser nulo"]
    }

}
```

Essa resposta será serializada para dentro da instância, no nosso caso, a instância do modelo `Pessoa`, em um atributo chamado `errors`. Para isso, vamos fazer uma pequena mudança no adaptador principal para que o `ember-data` entenda a mensagem de validação. Sobrescrevemos o método `handleResponse`:

```
//app/adapters/application.js
```

```
import DS from 'ember-data';

export default DS.RESTAdapter.extend({
  host: 'http://agenda.clairton.eti.br',
  namespace: null,
  coalesceFindRequests: true,
  handleResponse(status, headers, payload) {
    if (this.isInvalid(...arguments)) {
      payload.errors = DS.errorsHashToArray(payload.errors);
    }
    return this._super(...arguments);
  }
});
```

Podemos então iterar sobre esses erros mostrando para o usuário o que há de errado, como no exemplo seguinte:

```
{{#each pessoa.errors.nome as |error|}}

  <br/><div class='erro'>{{error.message}}</div>

{{/each}}
```

Adicionando esse trecho logo abaixo do campo `nome` no componente `formulario-pessoa` , teríamos algo assim quando tentarmos salvar a pessoa se preenchermos o nome:

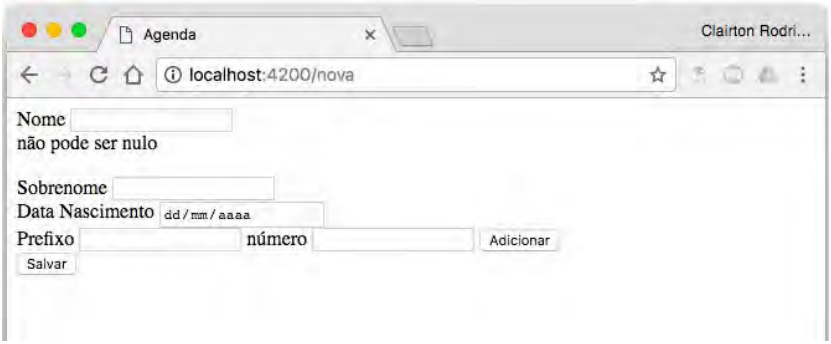

Figura 20.1: Mensagem de validação vinda do back-end

É interessante criarmos um componente para encapsular e reutilizar esse trecho de código, para assim não precisarmos de uma `each` a cada atributo. Então, vamos criar um componente chamado `mensagem-validacao`:

```
ember generate component mensagem-validacao
```

Precisamos que esse componente receba o atributo `erros` com os erros de validação, então o arquivo `app/templates/components/mensagem-validacao.hbs` deverá ficar assim:

```
{{!app/templates/components/mensagem-validacao.hbs}}

{{#each erros as |erro|}}

  <div class='erro'>{{erro.message}}</div><br/>

{{/each}}
```

Nesse componente, apenas esperamos receber um parâmetro `erros` com as violações vindas do back-end. Para o exemplo anterior, ficaria assim:

```
{{mensagem-validacao erros=pessoa.errors.nome}
```

Agora vamos usar esse componente recentemente criado no componente `formulario-pessoa`:

```
{{!app/templates/componets/formulario-pessoa.hbs}}

<label for="nome">Nome</label>
{{input id="nome" value=pessoa.nome action='salvarAoPressionarEnt
er' on='enter'}}
{{mensagem-validacao erros=pessoa.errors.nome}}
```

```
<br/>

<label for="sobrenome">Sobrenome</label>
{{input id="sobrenome" value=pessoa.sobrenome}}
{{mensagem-validacao erros=pessoa.errors.sobrenome}}
<br/>

<label for="nascidoEm">Data Nascimento</label>
{{input id="nascidoEm" value=pessoa.nascidoEm type='date'}}
{{mensagem-validacao erros=pessoa.errors.nascidoEm}}
<br/>

<label for="prefixo">Prefixo</label>
{{input id="prefixo" value=telefone.prefixo type='number'}}
<label for="numero">número</label>
{{input id="numero" value=telefone.numero type='number'}}
<button {{action 'adicionarTelefone' telefone}}>Adicionar</button>


<br/>

{{#each pessoa.telefones as |telefone|}}
    ({{telefone.prefixo}}) {{telefone.numero}}
    <button {{action 'removerTelefone' telefone}}>Remover</button>

    <br/>
{{/each}}


<button id="salvar" {{action 'salvar' pessoa}}>Salvar</button>
```

Dessa forma, o componente `mensagem-validacao` mostrará os erros na tela? Podemos simular isso por meio de um teste de integração. O teste de integração não é tão completo quanto o de aceitação, mas bem mais rápido. No teste de integração, passamos valores de entrada e testamos o resultado de saída.

Quando criamos o componente `mensagem-validacao` através do gerador do `ember-cli` , também foi criado um arquivo de teste em `tests/integration/components/mensagem-validacao-test.js` com um teste chamado `it renders` :

```
//tests/integration/components/mensagem-validacao-test.js

import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';

moduleForComponent('mensagem-validacao', 'Integration | Component
 | mensagem validacao', {
  integration: true
});

test('it renders', function(assert) {

  // Set any properties with this.set('myProperty', 'value');
  // Handle any actions with this.on('myAction', function(val) {
... });

  this.render(hbs`{{mensagem-validacao}}`);

  assert.equal(this.$().text().trim(), '');

  // Template block usage:
  this.render(hbs`
    {{#mensagem-validacao}}
      template block text
    {{/mensagem-validacao}}
  `);

  assert.equal(this.$().text().trim(), 'template block text');
});
```

Esse código simplesmente está testando se há algum erro de sintaxe. O interessante é que vemos como ele consegue renderizar o componente de forma isolada na linha:

```
this.render(hbs`{{mensagem-validacao}}`);
```

Podemos também setar variáveis no contexto e passá-las como parâmetro:

```
this.set('nomeDaPessoaObrigatorio', {errors:{nome:['Nome Obrigató
```

```
rio']}});

this.render(hbs`{{mensagem-validacao erros=nomeDaPessoaObrigatori
o}}`);
```

Também podemos testar se o HTML gerado é o esperado, nesse caso deveria ter um elemento `div` com a classe `erro`:

```
assert.equal(this.$('.erro').length, 1);
```

E dentro desse elemento HTML com a classe `erro`, deveria ter o conteúdo `Nome Obrigatório`:

```
assert.equal(this.$('.erro').text().trim(), 'Nome Obrigatório');
```

Vamos chamar esse teste de `Deveria mostrar uma mensagem`, substituindo o que já existe. O arquivo inteiro ficaria assim:

```
//tests/integration/components/mensagem-validacao-test.js

import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';

moduleForComponent('mensagem-validacao', 'Integration | Component
 | mensagem validacao', {
  integration: true
});

test('Deveria mostrar uma mensagem', function(assert) {
  this.set('nomeDaPessoaObrigatorio', {errors:{nome:['Nome Obriga
tório']}});

  this.render(hbs`{{mensagem-validacao erros=nomeDaPessoaObrigato
rio}}`);

  assert.equal(this.$('.erro').length, 1);
  assert.equal(this.$('.erro').text().trim(), 'Nome Obrigatório')
```

```
;
});
```

Porém, ao rodarmos esse teste visitando o endereço `http://localhost:4200/tests?filter=Deveria%20mostrar%20uma%20mensagem`, percebemos que o resultado ainda não é o esperado:

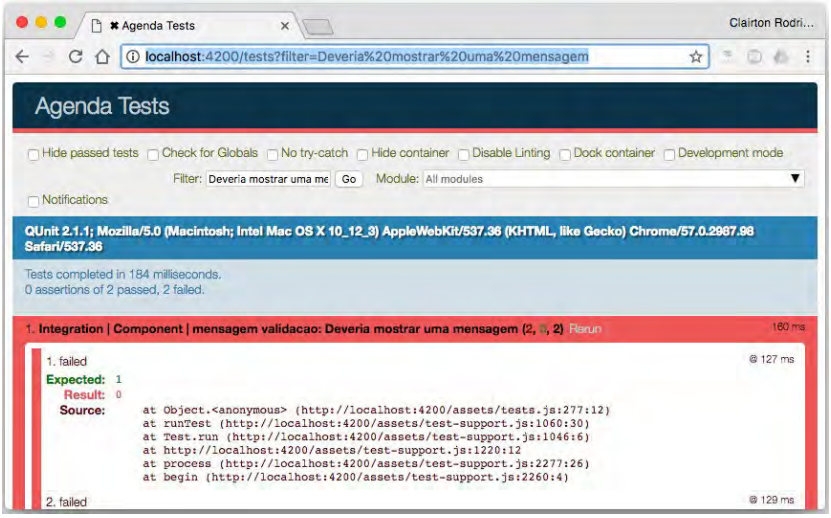

Figura 20.2: Falha ao mostrar uma mensagem

Isso acontece porque, quando o componente `mensagem-validacao` é usado no componente `formulario-pessoa`, passamos como parâmetro o atributo `errors` da instância do modelo `Pessoa`. Esse atributo é uma instância da classe do `ember-data`, nomeado `DS.Errors`, então precisamos importar essa classe:

```
import DS from 'ember-data';
```

Também devemos criar uma nova instância dela:

```
let erros = DS.Errors.create();
```

E então, adicionar uma mensagem para o atributo `nome`:

```
erros.add('nome', 'Nome Obrigatório');
```

E, por fim, setar o atributo `nomeDaPessoaObrigatorio` com a variável `erros`:

```
this.set('nomeDaPessoaObrigatorio', erros);
```

Então o arquivo de teste ficará assim:

```
//tests/integration/components/mensagem-validacao-test.js

import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import DS from 'ember-data';

moduleForComponent('mensagem-validacao', 'Integration | Component
 | mensagem validacao', {
  integration: true
});

test('Deveria mostrar uma mensagem', function(assert) {
  let erros = DS.Errors.create();
  erros.add('nome', 'Nome Obrigatório');

  this.set('nomeDaPessoaObrigatorio', erros);

  this.render(hbs`{{mensagem-validacao erros=nomeDaPessoaObrigato
rio}}`);

  assert.equal(this.$('.erro').length, 1);
  assert.equal(this.$('.erro').text().trim(), 'Nome Obrigatório')
;
});
```

Agora visitando o endereço `http://localhost:4200/tests?`

`filter=Deveria%20mostrar%20uma%20mensagem` , para rodar o teste, vimos que houve sucesso:

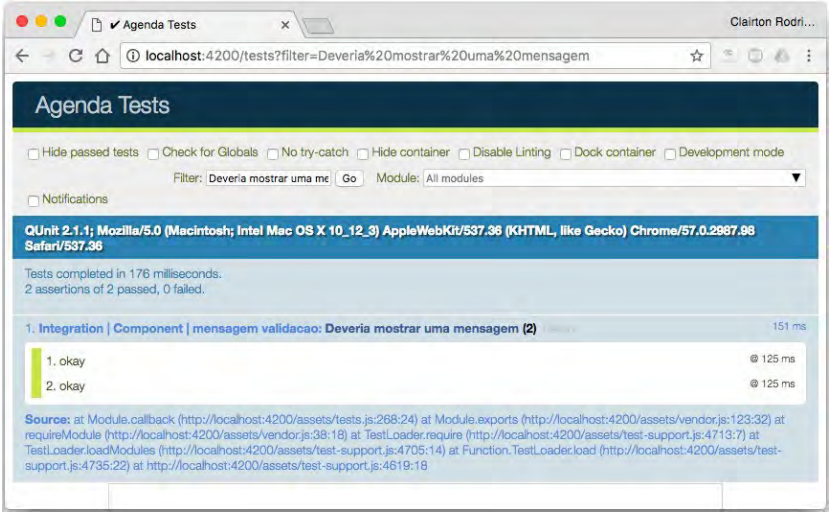
✔ Agenda Tests
Clairton Rodri...
localhost:4200/tests?filter=Deveria%20mostrar%20uma%20mensagem
Agenda Tests
Hide passed tests
Check for Globals
No try-catch
Hide container
Disable Linting
Dock container
Development mode
Filter: Deveria mostrar uma me
Go
Module: All modules
Notifications
QUnit 2.1.1; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.98 Safari/537.36
Tests completed in 176 milliseconds.
2 assertions of 2 passed, 0 failed.
1. Integration | Component | mensagem validacao: Deveria mostrar uma mensagem (2)
151 ms
1. okay
@ 125 ms
2. okay
@ 125 ms
Source: at Module.callback (http://localhost:4200/assets/tests.js:268:24) at Module.exports (http://localhost:4200/assets/vendor.js:123:32) at requireModule (http://localhost:4200/assets/vendor.js:38:18) at TestLoader.require (http://localhost:4200/assets/test-support.js:4713:7) at TestLoader.loadModules (http://localhost:4200/assets/test-support.js:4705:14) at Function.TestLoader.load (http://localhost:4200/assets/test-support.js:4735:22) at http://localhost:4200/assets/test-support.js:4619:18

Figura 20.3: Sucesso no teste ao validar mensagens

A aplicação que estamos escrevendo ainda está com o HTML bem arcaico, utilizando a tag `br` para indicar quebras de linhas. O ideal seria termos uma aplicação responsiva, que se adapte a vários dispositivos, seja um smartphone ou um computador de mesa. Esse é um ponto interessante do Ember. Como ele apenas usa tecnologia padrões da Web, podemos integrar bibliotecas já construídas para resolver problemas específicos - no caso, estamos falando do Twitter Bootstrap, o que abordaremos a seguir.

Capítulo 21

ADICIONANDO TWITTER BOOTSTRAP COMO DEPENDÊNCIA

Para gerenciar as dependências, o `ember-cli` usa principalmente o NPM (*Nodejs Package Manager*), e também o gerenciador de pacotes Bower. Porém, a tendência é que o Bower caia em desuso à medida que os pacotes forem portados para o NPM.

O Gerenciamento de Dependências não é novidade para o desenvolvimento Web. Na verdade, ele já é feito desde os primórdios, utilizando a tag `script` e setando a localização de arquivos através do atributo `src` . Vamos a um exemplo de adição do jQuery:

```
<script src="https://code.jquery.com/jquery-3.1.1.js" />
```

Para otimizar o carregamento da página, geralmente usávamos uma versão minificada para produção. Esses arquivos tinham quebras de linhas removidas, variáveis com nomes encurtados, tudo para ficar com o menor tamanho possível.

```
<script src="https://code.jquery.com/jquery-3.1.1.min.js" />
```

Entretanto, o controle dessas versões minificadas e não minificadas ficava difícil. Às vezes, um desenvolvedor acabava comitando a versão não minificada para produção, o que era percebido apenas dias depois. Ou acontecia um erro de JavaScript que necessitava de uma análise de contexto, dificultada pelo uso do arquivo minificado.

Essas e outras situações são parcialmente resolvidas com um gerenciamento de dependências. O Bower é usado para gerenciar essas dependências de front-end, e o NPM funciona apenas para módulos Node.js. As dependências se encontram dentro da tag `dependencies`, especificando nome e versão, assim como o NPM.

```
{
  "name": "agenda",
  "dependencies": {

  }
}
```

Novas dependências do Bower podem ser instaladas pelo comando:

```
bower install alguma-lib@versao --save
```

É importante não esquecer de usar o parâmetro `--save` para a que dependência seja persistida no arquivo `bower.json`. Já os pacotes NPM podem ser instalados por meio do comando:

```
npm install nome-do-pacote --save|--save-dev
```

É importante lembrar de utilizar o parâmetro `--save-dev` ou `--save` para que a dependência seja persistida no arquivo `package.json`, que armazena as dependências do projeto com suas respectivas versões. A diferença entre os dois parâmetros é que o `--save` leva a dependência junto quando o atual pacote for usado por um terceiro projeto.

Como mencionado no capítulo anterior, nossa missão neste capítulo é deixar nossa aplicação mais apresentável, e que também se adapte ao tipo de dispositivo que a esteja acessando, seja em um navegador instalado em um smartphone ou tablet, ou em uma televisão de 52". Para isso, vamos usar uma biblioteca especialista nisso, o Twitter Bootstrap. O primeiro passo é instalá-lo através do Bower:

```
bower install --save bootstrap
```

Vai demorar um pouquinho para terminar o processo, isso dependendo da sua conexão com a internet. Se visualizar o arquivo `bower.json` agora, verá a dependência nova instalada:

```
{
  "name": "agenda",
  "dependencies": {
    "bootstrap": "^3.3.7"
  }
}
```

Precisamos dizer para o `ember-cli` que, ao construir o projeto, ele precisa incluir os arquivos do Bootstrap. Para isso, nós o importamos no arquivo `ember-cli-build.js`. Este arquivo parametriza o processo de construção efetuado pelo Brocolli.js, usado pelo `ember-cli` para organizar o front-end.

Será necessário especificar tanto os arquivos JavaScript quanto as folhas de estilos. Esses arquivos foram colocados na pasta `bower_components` quando executamos o comando de instalação.

```
//ember-cli-build.js

var EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function(defaults) {
  var app = new EmberApp(defaults, {});

  app.import('bower_components/bootstrap/dist/css/bootstrap.css')
;
  app.import('bower_components/bootstrap/dist/css/bootstrap-theme
.css');
  app.import('bower_components/bootstrap/dist/js/bootstrap.js');

  return app.toTree();
};
```

Se você estiver rodando o comando `ember server`, será necessário que pare o processo e inicie novamente, para que as alterações sejam aplicadas.

Agora, já podemos utilizar as classes do Bootstrap. Inicialmente, para que haja a centralização do conteúdo, definiremos a classe `container` para a tag `body`:

```
<!--app/index.html-->
...
<body class="container">
...
```

Com isso, você já vai perceber que houve uma mudança no visual da lista de pessoas:

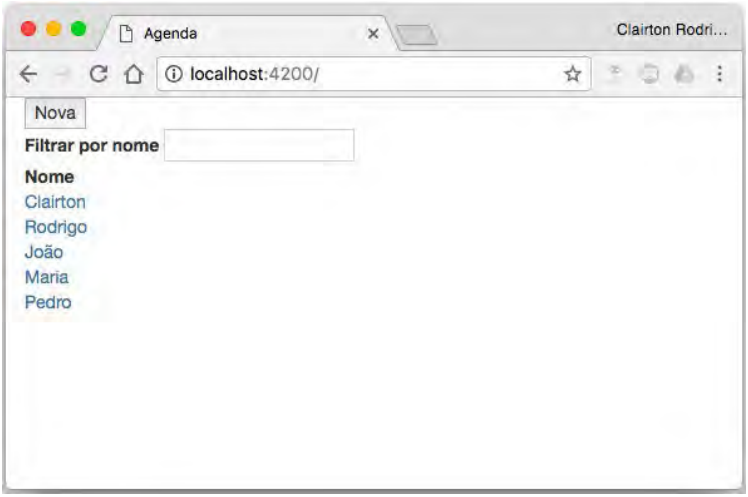

Figura 21.1: Somente adicionando a classe container

Agora, podemos estilizar os templates. Vamos começar colando a lista de pessoas em uma tabela estilizada, melhorando a visualização com linhas de cores diferentes. Para isso, usaremos as classes `table table-striped` :

```
{{!app/templates/lista.hbs}}

<button {{action 'nova'}}>Nova</button>
<br/>

<label for="filtro-nome">Filtrar por nome</label>
{{input id="filtro-nome" value=nome}}
<br/>

<table class="table table-striped">
  <thead>
    <tr>
      <th>Nome</th>
    </tr>
  </thead>
  <tbody>
```

```
    {{#each model as |pessoa|}}
      <tr>
        <td>
          {{#link-to 'pessoa' pessoa.id}}{{pessoa.nome}}{{/link-t
}}
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Como podemos ver, ficou bem mais fácil visualizar a separação dos registros:

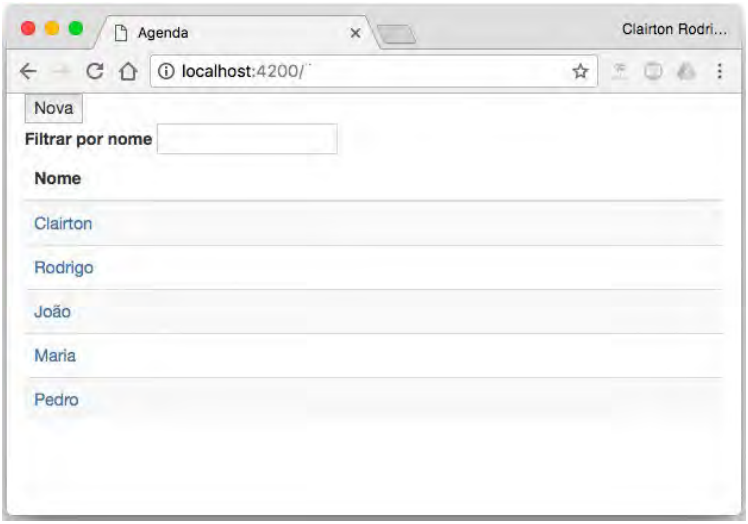

Figura 21.2: Adicionando uma tabela estilizada

Para adaptar a aplicação para dispositivos com vários tamanhos de tela, o Bootstrap utiliza o conceito *Grid System*, dividindo a tela em 12 colunas. Primeiro, faremos para dispositivos muito pequenos. Usaremos a classe `col-xs-8` para o campo de pesquisa e `col-sm-offset-1 col-xs-3` para o botão que

direciona para o novo. Isso quer dizer que, das 12 colunas, o campo de pesquisa terá 8 mais 1 coluna de intervalo e 3 colunas para o botão.

```
 {{!app/templates/lista}}

 ...

 <div class="row">
  <div class="col-xs-8">
    <label for="filtro-nome">Filtrar por nome</label>
    {{input id="filtro-nome" value=nome}}
  </div>
  <div class="col-xs-offset-1 col-xs-3">
    <button {{action 'nova'}} id="nova" type="button">Nova</butto
n>
  </div>
</div>

...
```

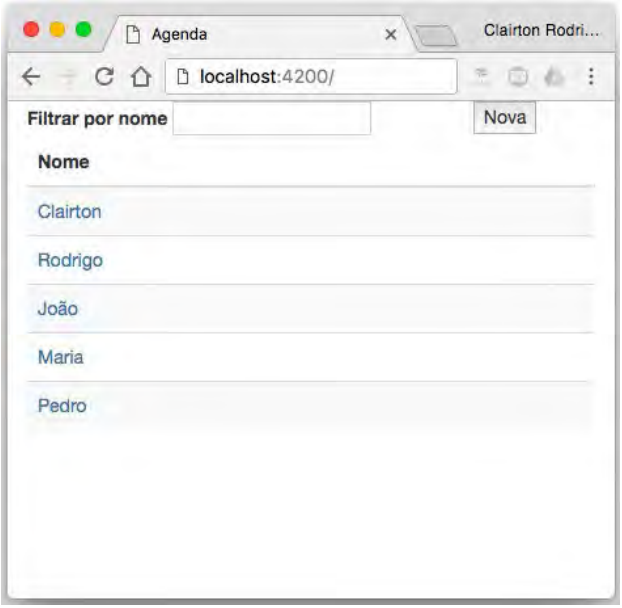

Figura 21.3: Exibição de lista de Pessoas em uma tela menor

Envolvemos tudo em uma `div` com a classe `row` para dizer que deve estar em uma linha, além de colocarmos os conteúdos do campo de pesquisa e do botão também envolvidos em uma `div` com as respectivas classes, para que agora possamos aplicar a classe `form-control` nesses elementos. Isso fará com que eles fiquem com um aspecto muito mais bonito.

```
{{!app/templates/lista}}

...

<div class="row">
 <div class="col-xs-8">
   <label for="filtro-nome">Filtrar por nome</label>
   {{input id="filtro-nome" class="form-control" value=nome}}
 </div>
```

```
  <div class="col-xs-offset-1 col-xs-3">
    <label></label>
    <button {{action 'nova'}} id="nova" class="form-control" type
="button">Nova</button>
  </div>
</div>

...
```

Para que o botão fique alinhado horizontalmente ao campo de pesquisa, atribuímos um rótulo vazio a ele. O resultado é excelente:

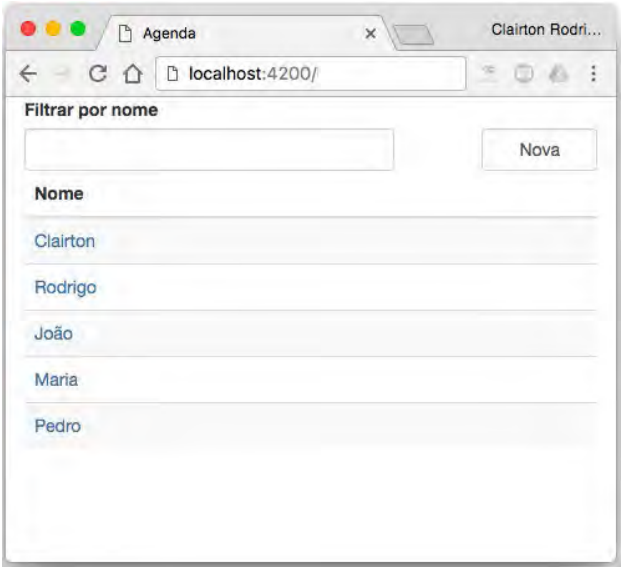

Figura 21.4: Alinhamento em tela pequena

Agora vamos preparar a tela para outros dispositivos. Até agora, preocupamo-nos apenas com as telas muito pequenas, representadas pela classe `col-xs-N`, em que `N` representa um número que pode ser de 1 a 12. Porém, também temos as telas pequenas com a classe `col-sm-N`, as médias com `col-md-N`, e as grandes com a classe `col-lg-N`.

Para as pequenas, vamos definir 8 colunas para a pesquisa, 2 de intervalo e 2 para o botão:

```
 {{!app/templates/lista}}

 ...

 <div class="row">
  <div class="col-xs-8 col-sm-8">
    <label for="filtro-nome">Filtrar por nome</label>
    {{input id="filtro-nome" class="form-control" value=nome}}
  </div>
  <div class="col-xs-offset-1 col-xs-3 col-sm-offset-2 col-sm-2">
    <label></label>
    <button {{action 'nova'}} id="nova" class="form-control" type
="button">Nova</button>
  </div>
</div>

...
```

Para dispositivos com tela média e grande, deixaremos 6 colunas para pesquisa, 4 de intervalo e duas para o botão.

```
{{!app/templates/lista.hbs}}

<div class="row">
  <div class="col-xs-8 col-sm-8 col-md-6 col-lg-6">
    <label for="filtro-nome">Filtrar por nome</label>
    {{input id="filtro-nome" class="form-control" value=nome}}
  </div>
  <div class="col-xs-offset-1 col-xs-3 col-sm-offset-2 col-sm-2 c
ol-md-offset-4 col-md-2 col-lg-offset-4 col-lg-2">
    <label></label>
    <button {{action 'nova'}} id="nova" class="form-control" type
="button">Nova</button>
  </div>
</div>

<table class="table table-striped">
  <thead>
```

```
    <tr>
      <th>Nome</th>
    </tr>
  </thead>
  <tbody>
    {{#each model as |pessoa|}}
      <tr>
        <td>
          {{#link-to 'pessoa' pessoa.id}}{{pessoa.nome}}{{/link-t
o}}
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Para a visualização da pessoa, faremos:

```
{{!app/components/mostra-pessoa.hbs}}

<div class="row">
    <label>Nome:</label> {{pessoa.nome}} {{pessoa.sobrenome}}
</div>

{{#if pessoa.nascidoEm}}
    <div class="col-xs-12 col-sm-6 col-md-3 col-md-2">
        <label>Aniversário:</label> {{formata-data pessoa.nascido
Em}}
    </div>
{{/if}}

<div class="row">
    <div class="col-xs-12 col-sm-6 col-md-3 col-md-2">
        <label>Telefones:</label>
    </div>
    {{#each pessoa.telefones as |telefone|}}
        <div class="col-xs-12 col-sm-6 col-md-3 col-md-2">
            ({{telefone.prefixo}}) {{telefone.numero}}
        </div>
    {{/each}}
</div>
```

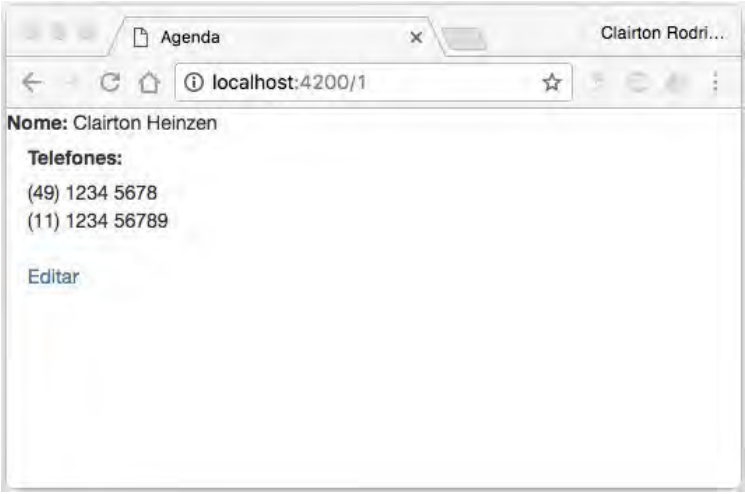

Figura 21.5: Mostrando Pessoa com estilização

O formulário usado para criar ou editar uma pessoa ficará assim:

```
{{!app/templates/componets/formulario-pessoa.hbs}}

<div class="col-xs-12 col-sm-12 col-md-4 col-lg-4">
    <label for="nome">Nome</label>
    {{input class="form-control" id="nome" value=pessoa.nome acti
on='salvarAoPressionarEnter' on='enter'}}
    {{mensagem-validacao erros=pessoa.errors.nome}}
</div>

<div class="col-xs-12 col-sm-12 col-md-4 col-lg-4">
    <label for="sobrenome">Sobrenome</label>
    {{input class="form-control" id="sobrenome" value=pessoa.sobr
enome}}
    {{mensagem-validacao erros=pessoa.errors.sobrenome}}
</div>

<div class="col-xs-12 col-sm-12 col-md-4 col-lg-4">
    <label for="nascidoEm">Data Nascimento</label>
    {{input class="form-control" id="nascidoEm" value=pessoa.nasc
```

```
idoEm type='date'}}
    {{mensagem-validacao erros=pessoa.errors.nascidoEm}}
</div>

<div class="row">
  <div class="col-xs-2 col-sm-2 col-md-2 col-lg-1">
    <label for="prefixo">Prefixo</label>
    {{input class="form-control" id="prefixo" value=telefone.pref
ixo type='number'}}
    {{mensagem-validacao erros=telefone.errors.prefixo}}
  </div>

  <div class="col-xs-7 col-sm-7 col-md-3 col-lg-2">
    <label for="numero">Número</label>
    {{input class="form-control" id="numero" value=telefone.numer
 type='number'}}
    {{mensagem-validacao erros=telefone.errors.numero}}
  </div>

  <div class="col-xs-3 col-sm-3 col-md-2 col-lg-2">
    <label></label>
    <button class="btn btn-default form-control" {{action 'adicio
narTelefone' telefone}}>
      Adicionar
    </button>
  </div>
</div>

{{#each pessoa.telefones as |telefone|}}
  <div class="row">
      <div class="col-xs-2 col-sm-2 col-md-2 col-lg-1">
      <label></label>
      {{input class="form-control" value=telefone.prefixo}}
    </div>

    <div class="col-xs-7 col-sm-7 col-md-3 col-lg-2">
      <label></label>
      {{input class="form-control" value=telefone.numero}}
    </div>
    <div class="col-xs-3 col-sm-3 col-md-2 col-lg-2">
      <label></label>
      <button class="btn btn-danger form-control" {{action 'remov
erTelefone' telefone}}>
        Remover
      </button>
```

```
    </div>
  </div>
{{/each}}

<div class="row">
  <div class="col-xs-offset-9 col-xs-3 col-sm-offset-9 col-sm-3 c
ol-md-offset-10 col-md-2 col-lg-offset-10 col-lg-2">
    <label></label>
    <button id="salvar" class="btn btn-primary form-control" {{ac
tion 'salvar' pessoa}}>
      Salvar
    </button>
  </div>
</div>
```

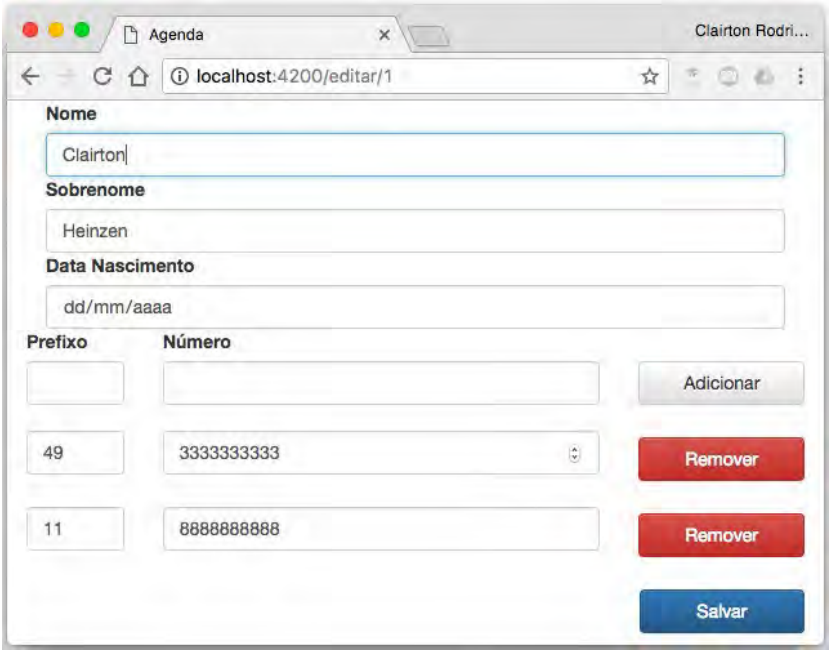

Figura 21.6: Estilizando o formulário

Caso a aplicação fosse acessada por uma tela maior, ficaria assim:

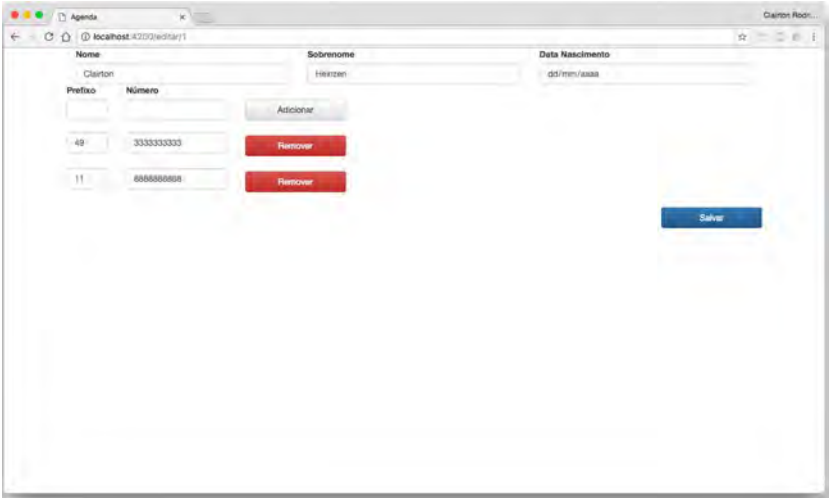

Figura 21.7: Visualização do formulário em uma tela maior

E por um smartphone, ficaria assim:

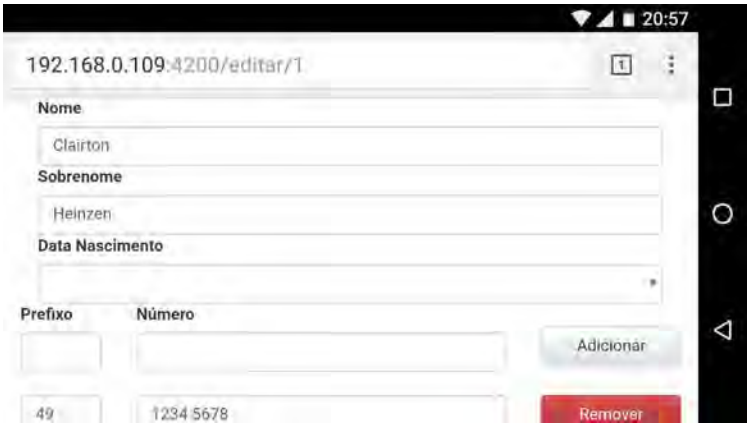

Figura 21.8: Visualização do formulário em uma tela menor

E, por último, mas não menos importante, formataremos as mensagens de validação:

```
{{!app/templates/components/mensagem-validacao.hbs}}

{{#each erros as |erro|}}

    <div class="erro alert alert-warning" role="alert">{{erro.mes
sage}}
        <button type="button" class="close" data-dismiss="alert"
aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
    </div>

{{/each}}
```

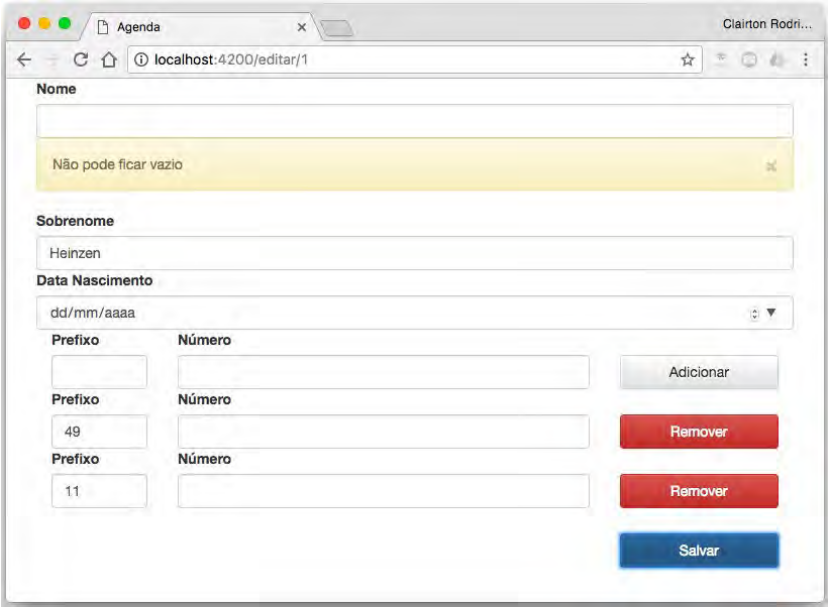

Figura 21.9: Mensagens de validação estilizadas

No início do capítulo, mencionamos sobre a minificação, mas não adicionamos os arquivos minificados no arquivo `ember-cli-build.js` . Essa etapa será estudada no próximo capítulo, no qual

falaremos sobre o processo de construção.

Capítulo 22

O PROCESSO DE CONSTRUÇÃO

Geralmente, temos configurações diferentes para desenvolvimento, testes e produção. Como já foi comentado, para ajudar no desenvolvimento, usamos as bibliotecas, sejam elas JavaScript, CSS ou HTML com o código formatado para que fique mais legível. Porém, quando o código for para a produção, devemos ter o menor tamanho possível.

Outro fator bastante importante é que geralmente temos o back-end rodando em nossa própria máquina no desenvolvimento, ou seja, nossa caixinha de areia, onde podemos testar tudo exaustivamente sem medo de danificar dados importantes. Também temos um back-end em um outro endereço que é usado para a produção.

O `ember-cli` possibilita configurarmos tantos ambientes quanto acharmos necessário, mas, por padrão, ele nos traz três ambientes: *development*, *test* e *production*. Eles estão especificados no arquivo `config/environment.js` .

```
/* jshint node: true */

module.exports = function(environment) {
```

```
  var ENV = {
    modulePrefix: 'agenda',
    environment: environment,
    rootURL: '/',
    locationType: 'auto',
    EmberENV: {
      FEATURES: {
        // Here you can enable experimental features on an ember
canary build
        // e.g. 'with-controller': true
      },
      EXTEND_PROTOTYPES: {
        // Prevent Ember Data from overriding Date.parse.
        Date: false
      }
    },

    APP: {
      // Here you can pass flags/options to your application inst
ance
      // when it is created
    }
  };

  if (environment === 'development') {
    // ENV.APP.LOG_RESOLVER = true;
    // ENV.APP.LOG_ACTIVE_GENERATION = true;
    // ENV.APP.LOG_TRANSITIONS = true;
    // ENV.APP.LOG_TRANSITIONS_INTERNAL = true;
    // ENV.APP.LOG_VIEW_LOOKUPS = true;
  }

  if (environment === 'test') {
    // Testem prefers this...
    ENV.locationType = 'none';

    // keep test console output quieter
    ENV.APP.LOG_ACTIVE_GENERATION = false;
    ENV.APP.LOG_VIEW_LOOKUPS = false;

    ENV.APP.rootElement = '#ember-testing';
  }

  if (environment === 'production') {
```

```
  }

  return ENV;
};
```

Por padrão, o endereço que o `ember-data` vai comunicar é o mesmo usado para acessar o navegador `http://localhost:4200`. E é nesse arquivo que configuraremos o endereço do back-end para `http://agenda.clairton.eti.br` no atributo `host`, e o atributo `namespace` para nulo, ficando assim:

```
/* jshint node: true */

module.exports = function(environment) {
  var ENV = {
    host: 'http://agenda.clairton.eti.br',
    namespace: null,
    modulePrefix: 'agenda',
    environment: environment,
    rootURL: '/',
    locationType: 'auto',
    EmberENV: {
      FEATURES: {
        // Here you can enable experimental features on an ember
canary build
        // e.g. 'with-controller': true
      },
      EXTEND_PROTOTYPES: {
        // Prevent Ember Data from overriding Date.parse.
        Date: false
      }
    },

    APP: {
      // Here you can pass flags/options to your application inst
ance
      // when it is created
    }
  };
```

```
  if (environment === 'development') {
    // ENV.APP.LOG_RESOLVER = true;
    // ENV.APP.LOG_ACTIVE_GENERATION = true;
    // ENV.APP.LOG_TRANSITIONS = true;
    // ENV.APP.LOG_TRANSITIONS_INTERNAL = true;
    // ENV.APP.LOG_VIEW_LOOKUPS = true;
  }

  if (environment === 'test') {
    // Testem prefers this...
    ENV.locationType = 'none';

    // keep test console output quieter
    ENV.APP.LOG_ACTIVE_GENERATION = false;
    ENV.APP.LOG_VIEW_LOOKUPS = false;

    ENV.APP.rootElement = '#ember-testing';
  }

  if (environment === 'production') {

  }

  return ENV;
};
```

Agora importaremos essas configurações no adaptador principal que está no arquivo `app/adapters/application.js` , substituindo o endereço que já havíamos colocado pela configuração:

```
//app/adapters/application.js

import DS from 'ember-data';

import Config from 'agenda/config/environment';

export default DS.RESTAdapter.extend({
  host: Config.host,
  namespace: Config.namespace,
  coalesceFindRequests: true,
```

```
  handleResponse(status, headers, payload) {
    if (this.isInvalid(...arguments)) {
      payload.errors = DS.errorsHashToArray(payload.errors);
    }
    return this._super(...arguments);
  }
});
```

A construção de uma aplicação escrita em Ember com o `ember-cli` é feita através do comando `ember build`. O ambiente pode ser definido através do parâmetro `--environment=ambiente_desejado`. Para construirmos a aplicação para enviar em produção, usaríamos o comando `ember build --environment=production`, sendo que o ambiente `production` minificará os arquivos.

Os arquivos gerados estarão na pasta `dist` e podem ser copiados para um servidor `http` com o Apache ou NGNIX sem problemas, ou podem ser empacotados conforme a tecnologia que você está utilizando. Executando o comando `tree` dentro da pasta `dist`, teremos algo parecido com:

```
.
├── assets
│   ├── agenda-39e01cb44069bdf327dbecee6e0bc26b.js
│   ├── agenda-d41d8cd98f00b204e9800998ecf8427e.css
│   ├── vendor-a79d3fd97be28feaee24d23647196791.js
│   └── vendor-d41d8cd98f00b204e9800998ecf8427e.css
├── crossdomain.xml
├── index.html
└── robots.txt
```

Veja que o nome dos arquivos contém um hash no nome. Isso ocorre no processo de construção para produção e serve para que, quando uma nova versão for gerada, a aplicação antiga não seja mantida em cache.

Copiando o arquivo `index.html` e a pasta `assets` com seu conteúdo, a aplicação estará funcionando. No processo de construção, os novos recursos das novas versões de ECMAScript, como as arrow functions, são transpilados para algo compatível com a implementação dos navegadores atuais. Veja um exemplo de transpilação da arrow function a seguir:

```
() => {
    console.log('sua lógica aqui');
}
```

Seria traduzido para:

```
function(){
    console.log('sua lógica aqui');
}
```

Os arquivos CSS, HTML e JavaScript de terceiros terão o nome iniciado por *vendor*, enquanto os arquivos da própria aplicação estarão iniciando com o nome dela própria.

No próximo capítulo, implementaremos um add-on, que é uma forma prática de compartilhar funcionalidades entre aplicações escritas com Ember.

Capítulo 23

ADD-ONS NO EMBER

Já ouvi dizer que desenvolvedor bom é desenvolvedor preguiçoso. Não no sentido de ser preguiçoso para aprender, mas de não fazer a mesma coisa duas vezes. Concordo com isso plenamente, e a comunidade que desenvolve o Ember também.

Lembra dos componentes que criamos? Poderíamos empacotar em um add-on e reutilizá-los em outros projetos Ember. O mesmo pode ser feito com os serviços, controladores, modelos etc.

Enquanto uma aplicação Ember é criada com o comando `ember new nome-do-projeto`, um add-on é criado com o comando `ember addon nome-do-addon`. Também vale lembrar que podemos usar bibliotecas que já existem. Como já utilizamos o Twitter Bootstrap, agora vamos usar o Moment.js, uma biblioteca muito usada e mantida pela JS Foundation, assim com o jQuery e outros projetos bem conhecidos.

A intenção é criar um add-on que tenha um transformador que serialize a data para o formato `yyyy-MM-dd`. Então, vamos criar um projeto chamado `ember-moment-transform`, e executar esse comando em uma pasta na qual você deseja que o novo projeto esteja localizado:

```
ember addon ember-moment-transform
```

O processo feito é o mesmo de quando criamos a aplicação da agenda, ou seja, criará a estrutura de pastas e baixará as dependências. Entrando na pasta criada:

```
cd ember-moment-transform
```

E listando as pastas da mesma forma que quando criamos o projeto agenda:

```
tree -d -L 2 -I 'node_modules|bower_components'
```

Temos a seguinte saída:

```
.
├── addon
├── app
├── config
├── tests
│   ├── dummy
│   ├── helpers
│   ├── integration
│   └── unit
└── vendor
```

Existem duas pastas que não estavam presentes no projeto da agenda, a pasta `addon` e a `dummy`. A primeira é onde estarão localizadas as implementações de controladores, serviços que estarão compartilhados para nós usarmos. Já na segunda, temos a opção de desenvolver uma aplicação que será usada como modelo de uso do add-on atual.

Agora vamos instalar a dependência que empacota o Moment.js, para então poder usá-lo. Para isso, execute o comando:

```
ember install ember-cli-moment-shim
```

Também criaremos o transformador, que chamaremos de `moment`:

```
ember generate transform moment
```

O Moment.js tem muitas funcionalidades, mas vamos utilizar basicamente o método `format` para formatar a data. Esse método espera um parâmetro que é a formatação desejada:

```
moment(new Date()).format('yyyy-MM-dd');
```

E o método `toDate`, que devolve um objeto `date`:

```
moment('2017-02-28', 'yyyy-MM-dd').toDate();
```

Para usar o `moment`, também precisamos importá-lo:

```
import moment from 'moment';
```

Nosso transformador ficaria assim:

```
//addon/transforms/moment.js

import DS from 'ember-data';
import moment from 'moment';

export default DS.Transform.extend({
  deserialize(serialized) {
    if (serialized) {
      return moment(serialized, 'yyyy-MM-dd').toDate();
    }
    return serialized;
  },

  serialize(deserialized) {
```

```
    if (deserialized) {
      return moment(deserialized).format('yyyy-MM-dd');
    }
    return deserialized;
  }
});
```

Para usá-lo no atributo `nascidoEm` do modelo `Pessoa`, faríamos:

```
nascidoEm: DS.attr('moment')
```

Mas isso limitaria o nosso transformador ao formato `yyyy-MM-dd`. Poderíamos passar um parâmetro dizendo qual o formato que esperamos:

```
nascidoEm: DS.attr('moment', {format: 'yyyy-MM-dd'})
```

E então recuperamos esses parâmetros no transformador:

```
//addon/transforms/moment.js

import DS from 'ember-data';
import moment from 'moment';

export default DS.Transform.extend({
  deserialize(serialized, options) {
    if (serialized) {
      return moment(serialized, options.format).toDate();
    }
    return serialized;
  },

  serialize(deserialized, options) {
    if (deserialized) {
      return moment(deserialized).format(options.format);
    }
    return deserialized;
  }
});
```

Podemos também adicionar a dependência `ember-cli-moment-shim` ao projeto que for usar o `ember-moment-transform`. Para isso, criaremos uma receita com o comando:

```
ember generate blueprint ember-moment-transform
```

Essa receita deve ter o mesmo nome do add-on, no caso, `ember-moment-transform`. Agora vamos implementar o método `afterInstall`, indicando que seja adicionada a dependência ao projeto:

```
//blueprints/ember-moment-transform/index.js

/*jshint node:true*/
module.exports = {
  description: 'Install dependencies',

  normalizeEntityName: function() {},

  afterInstall: function() {
    return this.addAddonToProject('ember-cli-moment-shim', '^3.0.
0');
  }
};
```

Agora já podemos publicar a primeira versão do nosso add-on. A sua publicação é feita através do gerenciador de pacotes NPM e, para publicar um pacote, é necessário que ele tenha um nome ainda não usado, além de você precisar possuir uma credencial para logar no sistema.

Tendo as credenciais, efetue o login utilizando o comando `npm adduser` e informando o solicitado. Após isso, é possível publicar nosso add-on através do `npm publish`:

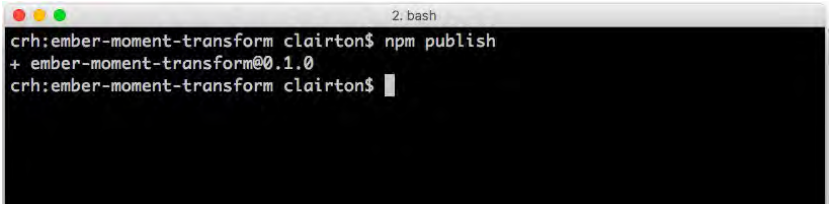

Figura 23.1: Publicando o add-on

Agora, para usá-lo em nossa aplicação da agenda, usaremos esse comando dentro do diretório do projeto agenda:

```
ember install ember-moment-transform
```

Ao olharmos o resultado, percebemos que durante o processo de instalação do `ember-moment-transform`, foi adicionado a dependência necessária chamada `ember-cli-moment-shim`:

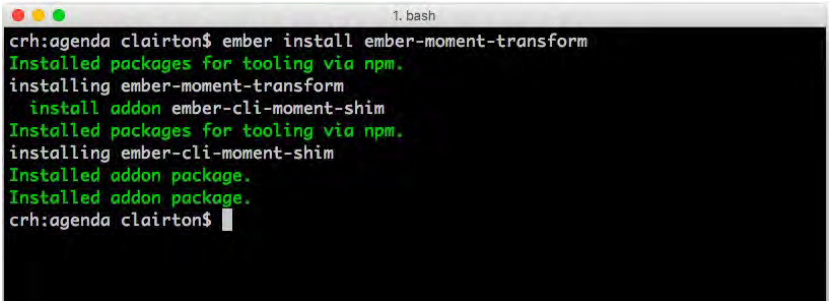

Figura 23.2: Instalando o add-on

Colocaremos duas novas linhas no arquivo `package.json`:

```
//package.json

...

    "ember-cli-moment-shim": "3.0.1",
    "ember-moment-transform": "0.1.0",
```

```
...
```

Então, basta usarmos o transformador no modelo `Pessoa`:

```
//app/models/pessoa.js

import DS from 'ember-data';

export default DS.Model.extend({
  nome: DS.attr('string'),
  sobrenome: DS.attr('string'),
  nascidoEm: DS.attr('moment', {format: 'yyyy-MM-dd'}),
  telefones: DS.hasMany('telefone', {async: true})
});
```

Agora podemos observar a forma da data ao salvarmos os dados de uma pessoa na ferramenta para desenvolvedor do navegador:

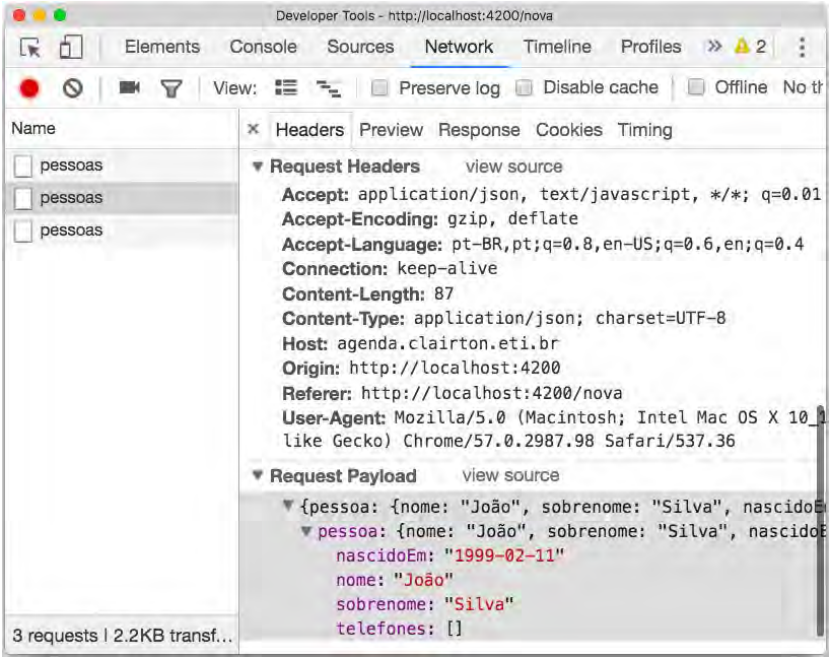

Figura 23.3: Requisição com data formatada

Em resumo, utilizamos uma biblioteca já escrita, usada e testada por muitos outros projetos, chamada Moment.js, criando uma pequena porção de código que poderá ser reusada em muitos projetos. No próximo capítulo, por sinal o último, vou indicar mais alguns lugares para você buscar ajuda com o Ember.js, caso necessite.

Capítulo 24

E AGORA, QUEM PODERÁ NOS DEFENDER?

Este livro abordou grande parte das características do Ember, dando uma boa base, mas ainda existe muita informação sobre o Ember na rede mundial de computadores. Temos desde o próprio site do projeto (https://emberjs.com) até vários artigos e tutoriais escritos pela comunidade, que por sinal, se mostra muito prestativa em compartilhar conhecimento e prestar auxílio se for solicitado.

Apesar de muitas tecnologias terem o guia definitivo, é difícil definir isso no Ember, pois, ainda que mantenha as bases, a evolução desse framework é contínua, prezando pela estabilidade sem estagnação.

Neste livro, abordamos a base do framework Ember.js, usando os geradores do `ember-cli` e conhecendo sua estrutura de pastas e organização. Também mapeamos o domínio da aplicação para modelos do `ember-data` , e depois persistimos e recuperamos dados com ele.

Utilizamos outras bibliotecas, como o Moment.js - escrito em JavaScript, e que tem uma gama de métodos para trabalhar com datas - e o Twitter Bootstrap - que agiliza o desenvolvimento de

aplicações responsivas utilizando o CSS. Além disso, criamos e publicamos add-ons para outros projetos escritos com o Ember.js. Outro assunto abordado foi a utilização do QUnit para construir testes automatizados.

Porém, se você tem algum problema ou dúvida que este livro não conseguiu esclarecer, com certeza a comunidade vibrante do Ember.js ficará satisfeita em poder lhe ajudar. Existe muita documentação no site do Ember (https://emberjs.com), além de vários artigos e tutoriais escritos por muitas pessoas.

Claro que você também é bem-vindo a entrar em contato no Fórum da Casa do Código, em http://forum.casadocodigo.com.br/, para tirar dúvidas e achar soluções.

Manter uma aplicação escrita com o Ember.js, atualizada com as mais novas funcionalidades e correções de erros, é muito fácil. Contando com versões estáveis de suporte longo lançadas a cada 6 meses, e com outros canais, como:

- https://emberjs.com/builds/canary/ - o canary, que tem as atualizações mais recentes;
- https://emberjs.com/builds/beta/ - o beta, onde é possível testar as novas features;
- https://emberjs.com/builds/release - o release, que conta com as alterações estáveis.

Quando alguma forma de usar o Ember.js está depreciada, ele emitirá uma mensagem orientando que essa forma de uso está desencorajada e que será removida em versões futuras. Isso possibilitou fazer a mudança do motor de renderização da versão 1 para 2, sem haver reescrita de aplicação ou quebra de código.

E-book gerado especialmente para Matheus Rodrigues F da Silva - matheusuniversal@yahoo.com.br

Com casos de sucesso em produtos conhecidos, como o Discourse, TravisCI, Zendesk e LinkedIn, além de empresas como Heroku, Yahoo! e Netflix, ele é uma ótima opção, pois traz as boas práticas e organização ergonômica do código.

Espero que este livro supra as necessidades de material que tive quando comecei a usá-lo, e que sua compilação ajude de alguma forma essa comunidade fantástica.